MICROPROCESSING FUNDAMENTALS

SEMINAR WORKBOOK

A SHORT COURSE FOR

SCIENTISTS AND ENGINEERS

BY

RAYMOND N. BENNETT

AND

DR. JOSEPH B. ROSS

AMERICAN INSTITUTE

FOR

PROFESSIONAL EDUCATION

CARNEGIE BUILDING
HILLCREST RD.
MADISON, N. J. 07940

PREFACE

As in learning to drive a car, a microprocessor must be practiced with. You cannot really learn how to use one from just reading books alone. This course includes a microcomputer and more information than can be covered in a three-day seminar; because it is the authors' purpose to give you sufficient background, written material, and hardware to be able to design a microcomputer system. BUT THIS CANNOT happen if the student does not study ALL the information given with the course and build up a system using the KIM-1.

COURSE OUTLINE 1-1

READING ASSIGNMENTS 2-1

EXPERIMENTS

- EXP. #1 3-1
- EXP. #2 3-3
- EXP. #3 3-6
- EXP. #4 3-8
- EXP. #5 3-9
- EXP. #6 3-10

LOGIC AND INTERFACE DEVICES

BASIC LOGIC 4.1-1
- I. NON-INVERTING BUFFER 4.1-2
- II. INVERTING BUFFER 4.1-2
- III. AND GATE 4.1-3
- IV. NAND GATE 4.1-4
- V. OR GATE 4.1-5
- VI. NOR GATE 4.1-6
- VII. EXCLUSIVE-OR GATE 4.1-7
- VIII. EXCLUSIVE-NOR GATE 4.1-8
- IX. DISCUSSION OF LOW-TRUE LOGIC 4.1-8

FLIP-FLOPS
- I. R-S LATCH 4.2-1
- II. R-S FLIP-FLOP 4.2-1
- III. D-TYPE FLIP-FLOP 4.2-2
- IV. J-K TYPE FLIP-FLOP 4.2-2
- V. T-TYPE FLIP-FLOP 4.2-2

DECODERS/DEMULTIPLEXERS 4.3-1

ENCODERS/MULTIPLEXERS 4.4-1

INTERFACE DEVICES
- OPEN-COLLECTOR LOGIC 4.5-1
- TRI-STATE LOGIC 4.5-2
- BUS TRANSCEIVERS 4.5-4

ANALYZING SOFTWARE PROBLEMS

- 5.0 THE SOFTWARE DESIGN PROCEDURE 5-1
- 5.1 STEP 1: DEFINE THE PROBLEM 5-2
- 5.2 STEP 2: PARTITION THE PROBLEM INTO FUNCTIONAL BLOCKS 5-9
- 5.3 STEP 3: ALGORITHM DEVELOPMENT FOR EACH PARTITION 5-12

ANALYZING SOFTWARE PROBLEMS (CONTINUED)

5.4 OBJECTIVES TO FLOWCHARTS 5-20

5.5 PROCEDURES AFTER ALGORITHM DEVELOPMENT 5-22

QUESTIONS 5-23

THE HARDWARE/SOFTWARE APPROACH TO MICROCOMPUTER DESIGN

INTRODUCTION 6-1

6.1 HARDWARE COST 6-2

6.1.1 SYSTEM SPEED 6-2
6.1.2 MEMORY REQUIREMENTS 6-3
6.1.3 I/O REQUIREMENTS 6-5
6.1.4 PERIPHERAL DEVICES 6-7
6.1.5 DEVICE SUPPORT 6-7
6.1.6 MICROPROCESSOR HARDWARE SELECTION SUMMARY 6-9

6.2 SOFTWARE COSTS 6-9

6.2.1 PROCESSOR ORGANIZATION 6-10
6.2.2 PROGRAM-STRUCTURE 6-11
6.2.3 IMPLEMENTATION LANGUAGE 6-11

6.3 SYSTEM COSTS 6-12

6.3.1 DEVELOPMENT COSTS 6-12
6.3.2 MODIFICATION COSTS 6-13
6.3.3 MAINTENANCE COSTS 6-14

6.4 A PERSPECTIVE ON COSTS 6-15

6.5 TRADING OFF SOFTWARE AND HARDWARE 6-16

6.5.1 CONDITIONS WHICH LEAD TO DESIGN TRADE OFFS 6-16
6.5.2 SYSTEM SPEED PROBLEMS 6-17
6.5.3 SYSTEM COST PROBLEMS 6-20

6.6 HARDWARE SPEED TRADE OFFS 6-21

6.6.1 PROCESSORS AND MEMORIES 6-21
6.6.2 DECODE LOGIC 6-22
6.6.3 MEMORY BUFFERS 6-22
6.6.4 SPECIALIZED INTERFACE DEVICES 6-23
6.6.5 INTERRUPTS 6-24

6.7 SOFTWARE TRADE OFFS 6-25

6.7.1 PROGRAM LOOPS AND SUBROUTINES 6-25

6.7.2 FUNCTIONAL COMPUTATIONS 6-26
6.7.3 REPEATED COMPUTATIONS 6-27

6.8 SUMMARY 6-27

REPRESENTING BINARY DATA 7-1

NUMBER SYSTEM CONVERSIONS

DECIMAL TO BINARY 8-1
BINARY TO DECIMAL 8-1
DECIMAL TO OCTAL 8-3
OCTAL TO DECIMAL 8-3
DECIMAL TO HEXADECIMAL 8-4
HEXADECIMAL TO DECIMAL 8-5
CONVERSIONS -
OCTAL TO BINARY, 8-5
HEXADECIMAL TO BINARY,
OCTAL TO HEXADECIMAL;
AND BACK

BCD NUMBERS 9-1

BINARY FRACTIONS 10-1

BINARY ARITHMETIC AND LOGIC INSTRUCTIONS

11.1 COMPUTER ARITHMETIC INSTRUCTIONS 11-1

11.1.1 TWO'S COMPLEMENT NOTATION 11-1
11.1.2 BINARY ARITHMETIC 11-3

11.2 COMPUTER LOGIC INSTRUCTIONS 11-5

11.2.2 LOGIC COMPLEMENT 11-5
11.2.2 LOGIC AND 11-6
11.2.3 LOGIC OR 11-6
11.2.4 LOGIC XOR 11-7

APPENDICES

A. MODIFIED 6500 OP CODE TABLE A-1

B. KIM INFORMATION

KIM PROGRAM DATA SHEET B-1

KIM BLOCK DIAGRAM B-2

KIM INTERFACING DATA SHEET B-3

KIM MONITOR IMPORTANT ADDRESSES B-4

C. COLLECTED KIM SOFTWARE

DISPLAY ROUTINE C-1
DIRECTORY C-2
VU TAPE C-3
SUPERTAPE C-4
TAPE DUPE C-7
MOVE-A-BLOCK C-8
HEX DUMP C-10
FREQUENCY COUNTER C-11
ANALOG TO DIGITAL CONVERSION DEMO C-13
REAL-TIME CLOCK C-14
TIMER C-15
HEDEC C-16
BINARY MULTIPLICATION AND DIVISION C-17
16 BIT SQUARE ROOT C-22
LUNAR LANDER C-23
HORSE RACE C-26
ONE-ARMED BANDIT C-27
KIMMAZE C-28
MUSIC MACHINE C-31
HUNT THE WUMPUS C-33

D. KIM DEMONSTRATION TAPE

INDEX D-1
PROGRAM HEX DUMPS D-2

E. SPECIAL APPLICATIONS

EIGHT BIT A TO D CONVERSION E-1
MULTICHANNEL ANALOG INTERFACE E-3

F. KIM/6500 INFORMATION SOURCES

KIM SOFTWARE SOURCES F-1
6500 MICROPROCESSOR SUPPLIERS F-3

G. GENERAL REFERENCE INFORMATION

H. TTL REFERENCE SHEETS

I. MOS TECHNOLOGY DATA SHEETS

J. MICROCOMPUTER BIBLIOGRAPHY

K. GLOSSARY OF COMMONLY USED TERMS

COURSE OUTLINE

MICROPROCESSING FUNDAMENTALS

COURSE OUTLINE

FIRST DAY

I. Introduction to Microprocessors and Microcomputers
   A. Hardware
   B. Software
   C. Number systems

II. Operating a Typical Microcomputer: The KIM-1
   A. Examining and modifying memory
   B. Loading and running sample programs
   C. Using the KIM audio cassette system
   D. Using the single step mode

III. Experiment 1: Loading and Running a Simple Program

IV. Microcomputer Architecture and Elementary Programming
   A. Simplified CPU model
   B. Data, address, and control buses
   C. Memory and I/O addressing
   D. The KIM monitor
   E. A selected subset of instructions

V. Programming Examples
   A. Parallel data input and output
   B. Use of the KIM-1 keyboard and display

VI. Experiment 2: Parallel Data Input and Output

SECOND DAY

I. Interfacing Microcomputers to External Devices
   A. Using programmable I/O lines for device control
   B. Device control software techniques
   C. Common interface devices
   D. Analog input and output techniques

II. Experiment 3: Controlling External Devices

III. Further Software
   A. Flags and conditional branches
   B. Counting and timing loops

THIRD DAY

I. Advanced Software
   A. Binary and decimal arithmetic
   B. Indexed addressing
   C. Indirect addressing

II. Interval Timers and Interrupts
   A. Using an interval timer for time delays
   B. The 6502 interrupt system
   C. Interval timer triggered interrupts
   D. Interrupt applications

III. Experiments 5 and 6: Using the Interval Timer and Interrupts

IV. Serial Data Input and Output
   A. The KIM-1 serial I/O system
   B. 20 mA current loop and RS-232 interfaces
   C. The ASCII code
   D. KIM monitor routines for serial I/O

IV. Further Topics as Requested

READING ASSIGNMENTS

READING ASSIGNMENTS

It is virtually impossible to read all the written material given with this course in the two nights during which the course is given. This material is given with the course to facilitate a higher level of expertise than can be presented or absorbed in a three-day seminar. The reading assignments listed below are highly recommended in order to receive the most from the next day's lecture. These assignments ONLY cover the two nights of the course. Read these assignments for information and understanding, NOT FOR DETAILED knowledge.

	KIM-1 USER MANUAL	HARDWARE MANUAL	PROGRAM MANUAL	SEMINAR WORKBOOK
FIRST NIGHT	Chapter 1 Sections 2.1 thru 2.4 Chapter 3 Section 4.1 Chapter 5	Sections 1.0 thru 1.2, 1.3.1, 1.3.3, 1.4 thru 1.4.1.2.4	Chapter 1 Chapter 2	Basic Logic Interface Devices
SECOND NIGHT		Sections 1.3.2 thru 1.3.2.6 1.5 thru 1.6.4.3	Chapters 3,4,5,6,7 Section 11.3, 11.3.1 Appendix H	Glossary of Common Terms

AFTER COMPLETION OF THE SEMINAR:

You should reread all the reading assignments FOR DETAILED KNOWLEDGE. There are many sections of the KIM-1 USER MANUAL, HARDWARE AND PROGRAM MANUALS, that were not made reading assignments. This DOES NOT MEAN that they are unimportant or not relevant. The reading assignments were made a basic understanding for the lecture material. You should read FOR DETAILED KNOWLEDGE the entire set of manuals and the SEMINAR WORKBOOK. You will find all the information in the WORKBOOK, highly condensed and extremely useful.

EXPERIMENTS

KIM EXPERIMENTS

WARNING! Your KIM-1 experimental set-up operates on low voltage only.

EXPERIMENT 1 Loading and Running a Simple Program

1. KIM-1 Initialization:
   Turn on 5V power. Press the RS key (reset). The display should light and show some random hex numbers.

2. Address Selection:
   Press AD to put KIM in address entry mode (address entry mode is automatically selected after reset). Enter 0000 on the keyboard. Observe the display see 0000 in the left four digits. You are looking at location 0000 in the KIM-1 read/write memory. The right two digits show the contents of this location. What are the contents? Look at the next location by pressing +. Continue pressing + to see what numbers are in your memory after system start up. Do you see a pattern to the numbers? Go to locations 0100, 0200, 0300, 0800, etc. and note the numbers you find. KIM read/write memory ranges from 0000 to 03FF. What numbers are found in locations where there is no physical memory?

3. Data Entry:
   Go to address 0000. Put KIM into the data entry mode by pressing the DA key. Press various keys and observe the display. To go to the next address, press +. For practice enter the following data into the KIM-1 memory:

address	data
0000	00
0001	01
0002	02
0003	03

   The + key allows you to increment the address in either the AD or DA mode. How do you go to a lower address or to a much higher address? You must return to the address mode and key in the new address then continue data entry in the DA mode.

4. Loading a Canned Program:
   Enter the program as listed on the coding sheet following this page. This program will cycle through the memory and display the contents of each location. For more information consult the program notes in your literature package.

PROGRAM: EXPERIMENT 1 - Display Routine

MICROPROCESSOR CODING SHEET

PAGE __ OF __

LABEL	ADDRESS	OP CODE	MNEMONIC	COMMENTS
	0000	A2		
	0001	04		
	0002	8A		
	0003	48		
	0004	A9		
	0005	62		
	0006	8D		
	0007	47		
	0008	17		
	0009	20		
	000A	19		
	000B	1F		
	000C	2C		
	000D	47		
	000E	17		
	000F	10		
	0010	F8		
	0011	68		
	0012	AA		
	0013	CA		
	0014	D0		
	0015	EC		
	0016	E6		
	0017	FA		
	0018	D0		
	0019	E6		
	001A	E6		
	001B	FB		
	001C	D0		
	001D	E2		

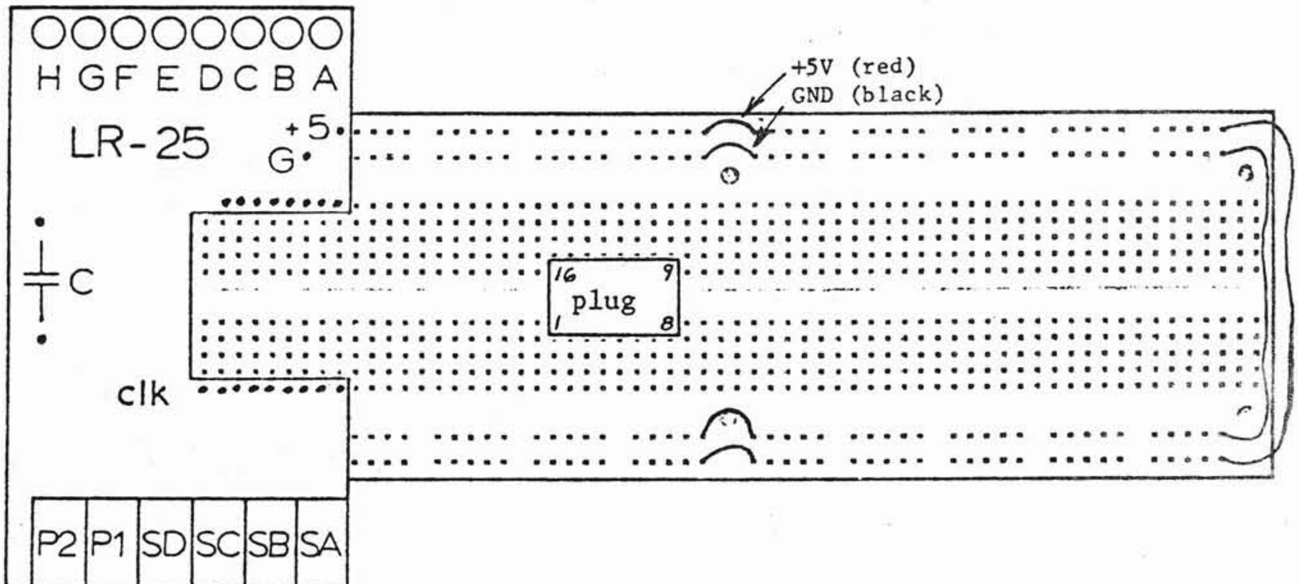
GENERAL COMMENTS:

5. Program Execution:
   Go to the beginning of the program using the AD mode (address 0000). Press the G key. The address display will count up and the data display will show the contents of each memory location. To stop the program and return to the KIM monitor, press RS. This program is written as a loop and will run forever.

6. Optional Experiment:
   Select one of the game programs in the literature package and load and run it. Lunar Lander, Horserace, and Kimaze are reasonably short.

EXPERIMENT 2 Parallel Data Input and Output

1. Prepare Experimental Equipment:
   Locate the SK-10 breadboarding socket and the LR-25 module. Plug the LR-25 into the SK-10 socket so that it is oriented as shown in the drawing below. Insert the flat cable plug in the center of the SK-10 as shown. The flat cable should enter the plug from the side away from you. This will put pin 1 on the front left side. Connect the black lead (GND) to the GND terminal on the LR-25. Connect the red lead (+5V) to the +5 LR-25 terminal.



Kim Experimental Breadboard

E+L Instruments
61 First St.
Derby, Conn 06418

Use 8 wires to connect the KIM PA output lines to the 8 LED indicators on the LR-25. Connect:

```
PA0 = pin 9  to IA
PA1 = pin 10 to IB
PA2 = pin 11 to IC
PA3 = pin 12 to ID
PA4 = pin 13 to IE
PA5 = pin 14 to IF
PA6 = pin 15 to IG
PA7 = pin 16 to IH
```

Use 6 wires to connect the KIM PB input/output lines to the switches and pulsers on the LR-25. Connect:

```
PB0 = pin 1  to SA
PB1 = pin 2  to SB
PB2 = pin 3  to SC
PB3 = pin 4  to SD
PB4 = pin 5  to P1 (0)
PB5 = pin 6  to P2 (0)
nc    pin 7  used in interrupt exp.
PB7   pin 8  to GND
```

2. Eight Bit Parallel Output:
   Establish the eight PA lines as OUTPUT LINES by storing the number $FF in the PA data direction register at location $1701. Use the KIM-1 keyboard to do this. Now use the KIM-1 keyboard to write various hex numbers into the output register and observe the effect on the 8 LED indicators. Go to address$1700 = PAD, press DA, then press hex keys. You will see the binary representation of the hex numbers shown in the data display.

   Note: The RS key resets the data direction registers to $00 = INPUT, so you must reenter the $FF in $1701 each time you use RS.

3. Parallel Input From External Switches:
   Establish PB0 - PB7 as INPUT by storing $00 in the PB data direction register at location $1703. Remember that this is done automatically by the RS key. Use the KIM-1 keyboard to look at the contents of the PB data register at location $1702. Operate the external switches and observe the effect on the memory contents.

4. Numerical Input from the Kim-1 Keyboard:
   The KIM-1 keyboard is scanned by a software routine. If no key is pressed the routine returns with $15 in the accumulator. If a key is pressed, the routine returns with the hex key code in the accumulator. The following program calls the keyboard input routine and transfers the contents of the accumulator to the PA output port. This will enable you to see the key codes on the 8 LED indicators.

Keyboard Input Test Program

```
0000  D8  CLD    set binary mode
0001  A2  LDX#   establish PA as out
0002  FF  $FF
0003  8E  STX@
0004  01  $01
0005  17  $17
0006  20  JSR@   call keyboard input routine
0007  6A  $6A
0008  1F  $1F
0009  8D  STA@   send contents of A out to PA
000A  00  $00
000B  17  $17
000C  4C  JMP@   loop back for more data
000D  06  $06
000E  00  $00
```

STA @ $00F9

To run this program, go to address $0000, then press G. The display will go dark because it is not used by this program.

5. Program Output to the KIM-1 Display:
The KIM-1 display is a software driven multiplexed seven segment display. We are going to use the display to output hex numbers. Three memory locations hold the numbers which are displayed by the display routine. The leftmost two digits are stored in $00FB, the middle two in $00FA, the right two in $00F9. To display a number, we must store it in the appropriate location and then call the display routine. If a continuous display is desired, you must include the call instruction in a loop so that it is repeatedly executed. The following program displays 010203 on the KIM-1 display.

Display Output Test Program

```
000F  A9  LDA#   load first number
0010  01  $01
0011  8D  STA@   store it in left display
0012  FB  $FB
0013  00  $00
0014  A9  LDA#   load second number
0015  02  $02
0016  8D  STA@   store it in middle display
0017  FA  $FA
0018  00  $00
0019  A9  LDA#   load third number
001A  03  $03
001B  8D  STA@   store it in right display
001C  F9  $F9
001D  00  $00
001E  20  JSR@   call display routine
001F  1F  $1F
0020  1F  $1F
0021  4C  JMP@   loop back to call routine again
0022  1E  $1E
0023  00  $00
```

FA

06

Note that this program starts at $000F. Go the program beginning and run the program. Press RS to stop the program.

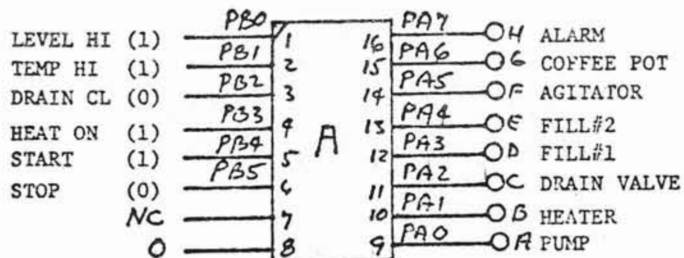
As a final project, you might like to link the keyboard entry program with the display output program so that the hex key codes are displayed in the right hand displays. How would the programs given need to be modified? Try it and see what you can do.

EXPERIMENT 3 Controlling External Devices

1. Single Step Execution of Programs:
The KIM single step function uses the $\overline{NMI}$ interrupt feature. In order to activate the single step function, you must load the proper address into the NMI vector locations. This is done by storing $00 in location $17FA and $1C in location $17FB. Once this vector has been loaded the ST key can be used to stop a program and return to the KIM monitor. You are now ready to try the single step function. Load a program and set the address to point to the program start location. Switch the keyboard switch to SSon. Press G and one instruction will be executed. While in the SS mode the data display will only show the first byte of each instruction. While in the SS mode, you can use the AD and DA modes to examine and modify any memory location. The PC key will recall the program counter value for the next instruction to be executed. After each instruction, the CPU registers are stored in memory where they can be examined or modified. This gives you the means of checking program execution or modifiying register values between steps. Memory locations for register storage are:

00EF	PCL	
00F0	PCH	
00F1	status register	(P)
00F2	stack pointer	(S)
00F3	accumulator	(A)
00F4	index register	(Y)
00F5	index register	(X)

2. We are now going to immagine that our KIM-1 is connected to an experimental apparatus. The devices to be controlled are hooked to the eight PA lines (used again as output). Of course we will have to use appropriate power drivers and interface devices to convert the TTL output signals to whatever is needed. We will also have several feedback signals to feed into our KIM-1. These are considered to be simple contact closures and are connected to the PB lines which will be programmed as inputs. The devices to be controlled and their input/output assignments are shown in the following schematic:



Let SA = Level, SB = Temp., SC = Drain, SD = Heat, P1 = Start, P2 = Stop.

3. We are now going to use the logical instructions OR and AND to turn individual devices on and off. Load the following program and single step through it so you can see the effect of each instruction on the output LED's which represent the actual devices. Note that you will have to look up your own op codes.

   Device Control Program

```
0000  A9  LDA#  $FF    establish PA as output
   2  8D  STA@  $1701
   5  A9  LDA#  $00    turn off all devices
   7  8D  STA@  $1700
   A  AD  LDA@  $1700  get output status
   D  09  ORA#  $40    turn on coffee pot
   F  8D  STA@  $1700
  12  AD  LDA@  $1700  get status
  15  09  ORA#  $01    turn on pump
  17  8D  STA@  $1700
  1A  AD  LDA@  $1702  get input status
  1D  29  AND#  $04    check state of drain valve
  1F  D0  BNEr  $F9    if drain is open, loop back and check again
  21  AD  LDA@  $1700  get status
  24  09  ORA#  $08    drain is closed so start fill #1
  26  8D  STA@  $1700
  29  AD  LDA@  $1700  get status
  2C  29  AND#  $F7    turn off fill #1
  2E  8D  STA@  $1700
      4C  JMP@  $1C4F
          etc.
```

As you run through the program, turn switch SC on and off to simulate having the drain valve open and closed.

Program termination:

If you want to have a program run just once, you must end it with a command to return to the KIM monitor. This can be done by terminating your program with: JMP@ $1C4F.

EXPERIMENT 4 Counting and Timing Loops

1. Counting Loops:
   The following example shows how to set up a counter (here the X register) to allow execution of a program segment for some preselected number of times. We could just as easily used the accumulator, the Y register or any r/w memory location as a counter.

   Counting Loop Program Example

```
               LDA#  $FF   establish PA as output
               STA@  $1701
               LDA#  $00   turn off all LED's
               STA@  $1700
      COUNT    LDX#  $0A   load counter with 10(10)
      LOOP     INC@  $1700 increment the output port
               DEX         decrement the counter
               BNEr  LOOP  if counter not zero, jump to loop
      DONE     JMP@  $1C4F return to the monitor
```

   Run this program in the SS mode and at full speed. Change the count value and observe the result.

2. Timing Loops:
   All operations in the KIM-1 system are timed by the crystal clock oscillator operating at a nominal 1.000 MHz. The oscillator is quite stable, but may not be exactly 1 MHz since that would require a more expensive crystal. If you need precise timing, check your oscillator with a good frequency counter. Each instruction requires a specific number of clock cycles for its execution. Thus program segments and loops can be used to produce very precise time delays which are as stable as the crystal clock. The number of cycles for each instruction is found on the MCS6500 Summary card and in the MOS Microcomputer Programming Manual. The following program yields a delay of 502 cycles = 502 microseconds from a single loop.

```
   Time Delay Program             cycles

               LDX#  $64            2
      LOOP     DEX                  2
               BNEr  LOOP           3
```

   The loop is 5 cycles and is executed 100 times. The initial LDX# adds the last 2 cycles. To obtain long delays, loops can be nested to produce delays of any length. Now that you have the basic idea here is a more complicated program. We put the time delay in a subroutine so that it can be readily used by other programs. The main program clears A then increments it and outputs it to the PA port. Each cycle is delayed by the time delay subroutine. You will have to look up the op codes. Start the main program at 0000 and the delay subroutine at 0013.

Time Delay Test Program with Subroutine

```
START   LDA#   $FF
        STA@   $1701
        LDA#   $00      clear A
SHOW    STA@   $1700    look at A
        CLC             clear carry before add
        ADC#   $01      add 1 to A
        JSR@   DELAY    delay 0.1 sec
        JMP@   SHOW     loop back to SHOW
DELAY   LDY#   $C8      load 200_10 into Y = T_y
LOOPY   LDX#   $62      load 98_10  into X = T_x
        STXz   $F5      waste 3 cycles
LOOPX   DEX             decrement X
        BNEr   LOOPX    if X not zero, loopx
        DEY             decrement Y
        BNEr   LOOPY    if Y not zero, loopy
        RTS             return
```

The total time delay here is $TD = 5T_y(T_x + 2) + 14$ microsec.

Run the program and try different values for T_y and T_x. You might try to write a program that would allow you to enter time constants from the keyboard in real time as the program is running.

This is a good program to use to see the effects of some of the other accumulator instructions. Replace the CLC, ADC# sequence with SEC, SBC#, or RORa, ROLa, ASLa, LSRa. If you replace a two byte instruction with a one byte instruction, be sure to add a NOP to fill the gap.

EXPERIMENT 5 The Interval Timer

1. The KIM-1 interval timer can produce a wide range of programmable time delays from a few microseconds to 250 mSEC. The interval timer consists of an eight bit down counter and a programmable clock divider which produces time intervals of 1 uSEC, 8 uSEC, 64 uSEC, or 1024 uSEC. The number of counts and the count interval are easily controlled. In this experiment we shall use the interval timer to produce a time delay subroutine. You should use the same main program used in EXP. 4 to test this routine. Start with the TDLY address = $1707, then try the other values shown in the following table:

TDLY	T_{int}	(X)	Delay
$1704	1 uSEC	$64	100 uSEC
$1705	8 uSEC	$64	800 uSEC
$1706	64 uSEC	$64	6400 uSEC
$1707	1 mSEC	$64	100 mSEC

Interval Timer Subroutine

```
INTDLY  PHA               save the contents of A
        LDX#   $64        load count
        STX@   TDLY       load counter and set divide ratio
WAIT    LDA@   $1707      get timer status
        BEQr   WAIT       if status = 0, wait
DONE    PLA               restore accumulator
        RTS               return
```

Note that the interval timer always runs in real time. If you single step through a program contining an interval timer delay, the program will flow right through the delay and not get hung up for N loops as is the case with timing loops.

EXPERIMENT 6 Interrupts

1. The interval timer can be programmed to interrupt the KIM-1 system every nnn machine cycles. In this experiment we are going to generate an interrupt every 0.2 sec and use this interrupt to run a program which will increment the PA output port. You should run a main program which does not use the PA port. The game programs, or the display routine used in experiment 1 are good for this purpose. Here is the interrupt routine:

Interval Timer Interrupt Program

```
1780  PHA                save A
1781  LDA#   $C8         load A with 200₁₀
1783  STA@   $170F       load timer and set divide ratio to 1024
1786  LDA#   $FF
1788  STA@   $1701       set PA to out
178B  INC@   $1700       increment PA lines
178E  PLA                restore original A
                         enable interrupt
178F  RTI                return from interrupt
```

We put this program in one of the small blocks of r/w memory not used by most programs. Set the IRQ interrupt vector to point to the above routine by storing the entry address in $17FE and $17FF (store $80 in $17FE and $17 in $17FF). You must connect the the interval timer output signal to the IRQ input line. This is accomplished as follows. Attach a spare 22 pin edge connector to the expansion lines on the KIM-1 board. Connect the orange clip to the IRQ input (pin 4). Connect a short jumper between pins 7 and 8 of the dip plug. Make sure PB7 is programmed as an input line even though it is used to send the timer signal out to IRQ. After a system reset (RS key used) you must enable the interval timer interrupt capability by reading location $170E once. This can be done manually using the KIM keyboard. You are now ready to run your main program. You should observe normal program execution and apparently simultaneous incrementing of the PA output indicators. Be sure the processor starts with the interrupt enabled by storing $00 in location $00F1 before running the program. THIS PROGRAM WILL NOT FUNCTION UNLESS YOU REMOVE THE GROUND WIRE FROM PIN # 8 ON THE 16-PIN RIBBON CONNECTOR.

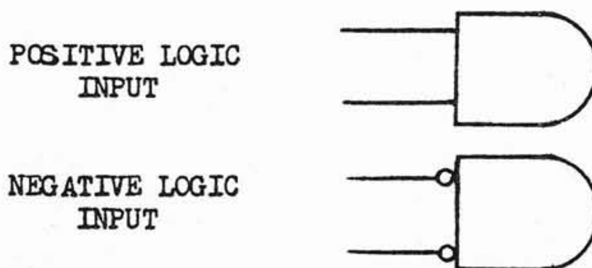
2. Optional Experiment: Frequency Counter
   Look up the Frequency Counter program in your literature package. Load it, connect PB7 to IRQ (jumper pins 7 and 8 and hook the orange clip to expansion connector pin 4 as done in exp. 6-1). Connect the LR-25 clock oscillator output to the counter input PB0 (pin 1). Run the program starting at $0000. Vary the oscillator frequency by inserting different sizes of capacitors in the terminals marked "C".

LOGIC AND INTERFACE DEVICES

BASIC LOGIC DEVICES

Although microprocessors are called (and often are used as) logic replacements, basic logic gates are still needed in most microcomputer systems. They are used for Buffers, Latches, Address Decoders, and Signal Conditioners. Therefore, it is important to have a good understanding and working knowledge of basic logic gates.

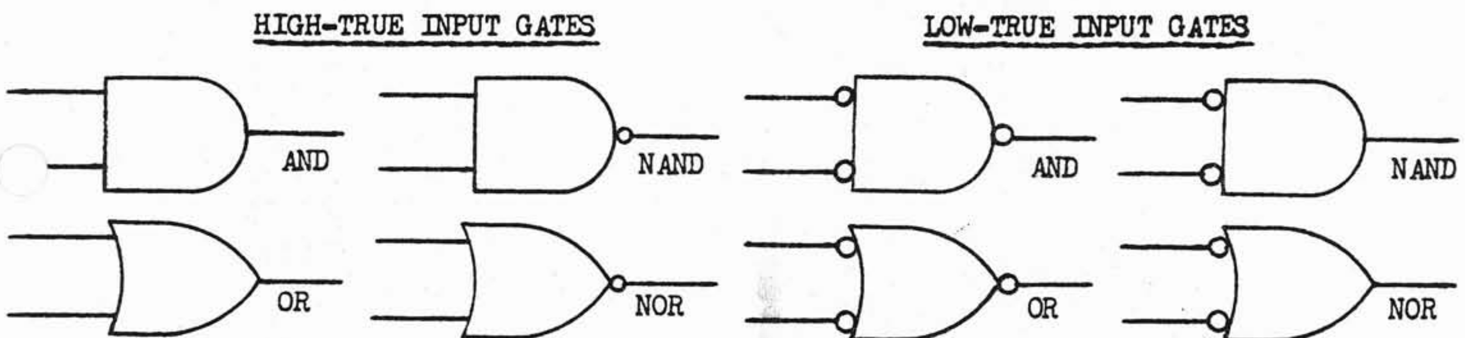
Digital logic operates in the binary number system. Therefore, any one input or output can only be in one of two distinct states, either a "1" or a "Ø". Normally, references made in regard to a digital signal, a logical "1" is greater than 2.0 volts and a logical "Ø" is less than 0.8 volts; this is called HIGH-TRUE or POSITIVE-TRUE LOGIC. LOW-TRUE or NEGATIVE-TRUE LOGIC is the opposite, a logical "1" is less than 0.8 volts and logical "Ø" is greater than 2.0 volts. On logical diagrams, the type of logic (Positive or Negative) is shown by the use of a circle in the input/output lead touching the logic symbol for the gate to indicate a LOW-TRUE input/output. The absence of this circle indicates a HIGH-TRUE input/output.



When the circle is used in an output lead of a POSITIVE-TRUE input gate or the absence of it in a LOW-TRUE input gate, it changes the name of the gate by adding the letter "N" in front of the gate's name, such as:

HIGH-TRUE INPUT GATES

LOW-TRUE INPUT GATES

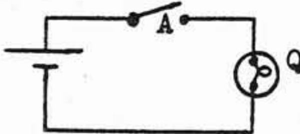


An explanation of the basic logic gates follows:

I. NON-INVERTING BUFFER

This device is used primarily to increase the load handling capabilities of another device. The output of this device will always be the same logic level as its input.

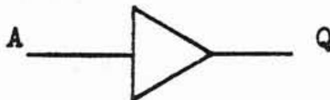
ANALOGY:



Closing the switch turns the lamp ON. Opening the switch turns the lamp OFF.

The switch closed represents a high input
The switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

LOGIC SYMBOL:



A - Input
Q - Output

TRUTH TABLE:

A	Q
1	1
0	0

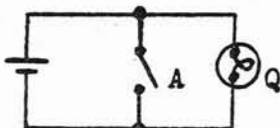
BOOLEAN EQUATION:

$$A = Q$$

II. INVERTING BUFFER

This device is used primarily for logic level inversion. The output of this device will always be the opposite logic level to its input.

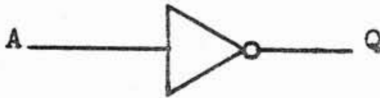
ANALOGY:



Closing the switch will short out the lamp and turn it off. Opening the switch will remove the short and turn on the lamp.

The switch closed represents a high input
The switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

LOGIC SYMBOL:



A - Input
Q - Output

The small circle at the end of the gate indicates output inversion.

TRUTH TABLE:

A	Q
1	0
0	1

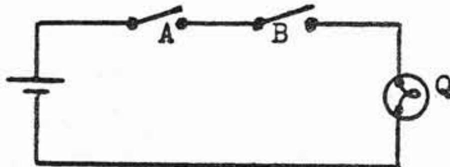
BOOLEAN EQUATION:

$$A = \overline{Q}$$

III. AND

This device used primarily to indicate whether or not all of its inputs are high at the same time. The output is HIGH-TRUE.

ANALOGY:



Both switches must be closed to turn the lamp on. If either or both switches are open, the lamp will be off.

A switch closed represents a high input
A switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

LOGIC SYMBOL:

A & B - INPUTS

Q - OUTPUT

TRUTH TABLE:

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

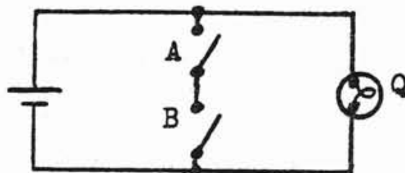
BOOLEAN EQUATION:

$$A \cdot B = Q$$

IV. NAND

This device is used the same as the AND, except the output is LOW-TRUE.

ANALOGY:



A switch closed represents a high input
A switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

When both switches are closed, they short out the lamp and turn it off. If either or both switches open, the short will be removed and turn the light on.

LOGIC SYMBOL:

A & B - Inputs

Q - Output

TRUTH TABLE:

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

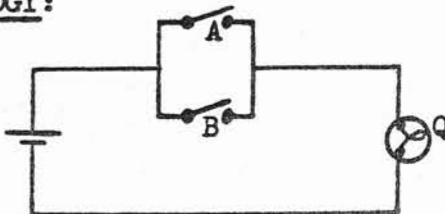
BOOLEAN EQUATION:

$$A \cdot B = \overline{Q}$$

V. OR

This device is used to indicate when at least one of its inputs is high. The output is HIGH-TRUE.

ANALOGY:



A switch closed represents a high input
A switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

Closing of either or both switches turns the light on. All the switches must be open to turn the lamp off.

LOGIC SYMBOL:

A & B - Inputs

Q - Output

TRUTH TABLE:

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

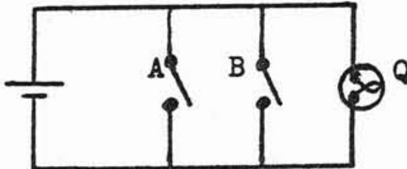
BOOLEAN EQUATION:

$$A + B = Q$$

VI. NOR

This device is used for the same purpose as the OR gate, except the output is LOW-TRUE.

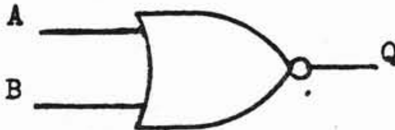
ANALOGY:



A switch closed represents a high input
A switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

Closing of either or both switches shorts out the lamp and turns it off. Both switches must be open to turn on the lamp.

LOGIC SYMBOL:



A & B - Inputs

Q - Output

TRUTH TABLE:

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

BOOLEAN EQUATION:

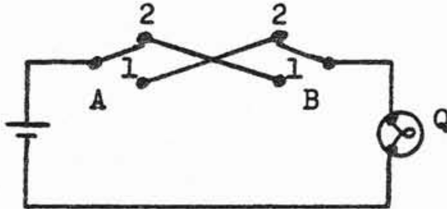
$$A + B = \overline{Q}$$

VII. EXCLUSIVE - OR

This device is used to indicate when one, and only one, input is high.

The output is HIGH-TRUE.

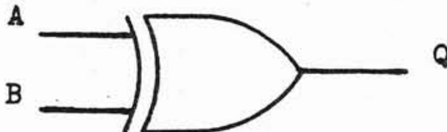
ANALOGY:



A switch in the "1" position represents a high input
A switch in the "2" position represents a low input
The lamp on represents a high output
The lamp off represents a low output

For the lamp to be on, one switch must be in the "1" position and one must be in the "2" position. Otherwise, the lamp will be off.

LOGIC SYMBOL:



Q

A & B - Inputs

Q - Output

TRUTH TABLE:

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

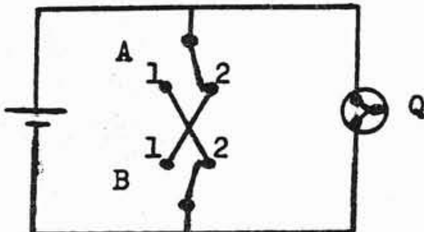
BOOLEAN EQUATION:

$$A \oplus B = A\overline{B} + \overline{A}B = Q$$

VIII. EXCLUSIVE-NOR

This device is the same as the EXCLUSIVE-OR gate, except the output is LOW-TRUE.

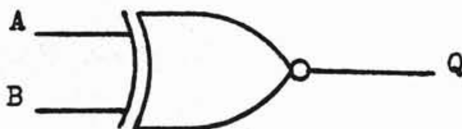
ANALOGY:



A switch in the "1" position represents a high input
A switch in the "2" position represents a low input
The lamp on represents a high output
The lamp off represents a low output

The only way to short-out the lamp and turn it off, is to have one switch in the "1" position and one switch in the "2" position. Otherwise, the lamp will be on.

LOGIC SYMBOL:



A & B - Inputs

Q - Output

TRUTH TABLE:

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

BOOLEAN EQUATION:

$$A \oplus B = A\overline{B} + \overline{A}B = \overline{Q}$$

IX. DISSCUSSION OF LOW-TRUE LOGIC

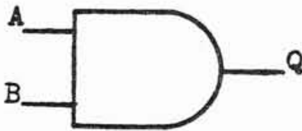
The preceeding discussion on the basic logic gates has not discussed gates with LOW or NEGATIVE-TRUE inputs. This is because there are no I.C.'s specifically designated for LOW-TRUE inputs, but a close examination

of the truth tables shows the following relationships:

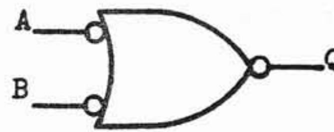
NOTE: In the following truth tables, "L" & "H" are used instead of "1" & "0" to reduce the confusion of what is a LOGICAL "1" or "0" between HIGH-TRUE and LOW-TRUE input LOGIC. An L ≤ 0.8 volts and an H ≥ 2.0 volts.

HIGH-TRUE INPUT = LOW-TRUE INPUT

HIGH-TRUE AND = LOW-TRUE OR



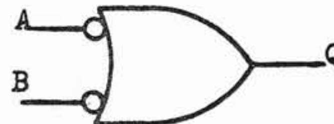
A	B	Q
L	L	L
L	H	L
H	L	L
H	H	H



A	B	Q
L	L	L
L	H	L
H	L	L
H	H	H

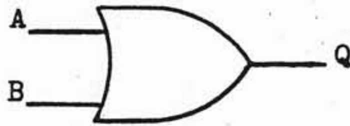
HIGH-TRUE NAND = LOW-TRUE NOR

A	B	Q
L	L	H
L	H	H
H	L	H
H	H	L

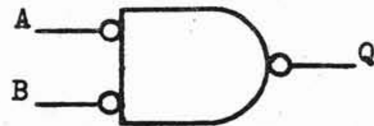


A	B	Q
L	L	H
L	H	H
H	L	H
H	H	L

HIGH-TRUE OR = LOW-TRUE AND

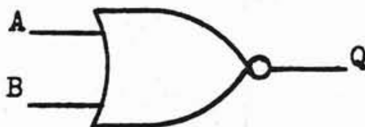


A	B	Q
L	L	L
L	H	H
H	L	H
H	H	H

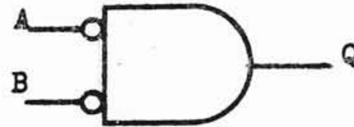


A	B	Q
L	L	L
L	H	H
H	L	H
H	H	H

HIGH-TRUE NOR = LOW-TRUE NAND



A	B	Q
L	L	H
L	H	L
H	L	L
H	H	L

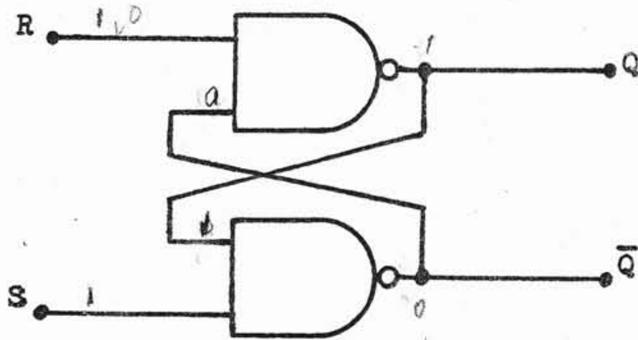


A	B	Q
L	L	H
L	H	L
H	L	L
H	H	L

FLIP-FLOPS

I. R-S LATCH:

The R-S latch was probably the first type of flip-flop ever built, (R = Reset & S = Set).



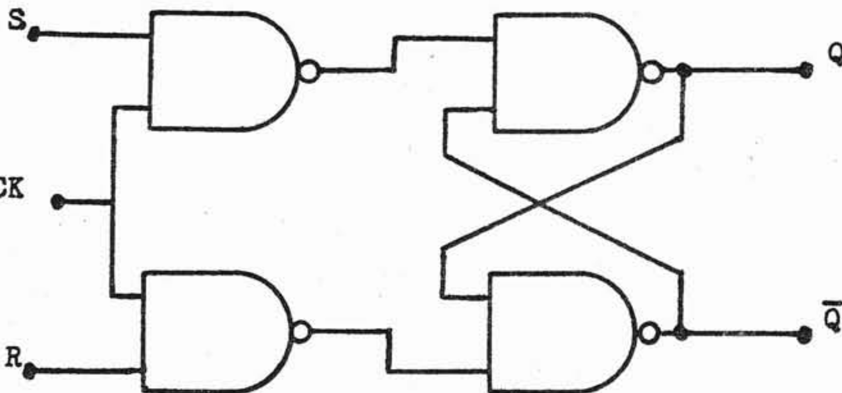
R	S	Q	$\overline{Q}$
0	0	1*	1*
0	1	1	0
1	0	0	1
1	1	NO CHANGE	

*Not allowed

To make the R-S latch into a clocked flip-flop, a clock input must be added.

II. R-S FLIP FLOP:

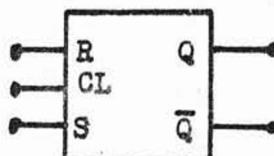
The R-S flip-flop is the simplest of the flip-flops.



R	S	Q	$\overline{Q}$
0	0	NO CHANGE	
0	1	1	0
1	0	0	1
1	1	1*	1*

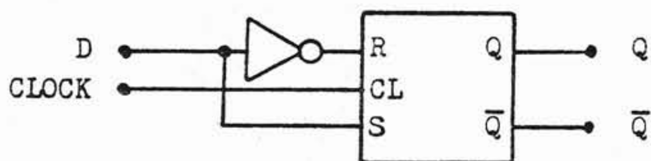
*Not allowed

The addition of the two NAND gates with the clock input changes it into a clocked R-S flip-flop. The inputs (R & S) can only change the outputs (Q & $\overline{Q}$) during a high input clock pulse. The R-S flip-flop is usually drawn in this manner:



III. DATA OR D-TYPE FLIP-FLOP:

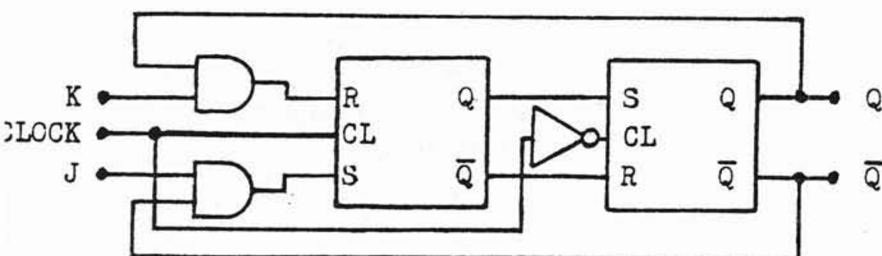
The D-Type F. F. is used primarily for a data latch. It can be made effectively from an R-S F. F. by:



D	Q	$\overline{Q}$
1	1	0
0	0	1

IV. J-K TYPE FLIP-FLOP:

The J-K or Master-Slave F. F. is used whenever data is to be trapped and latched at a given instant in time, such as in shift registers. It can be effectively made from two R-S F. F.'s by:

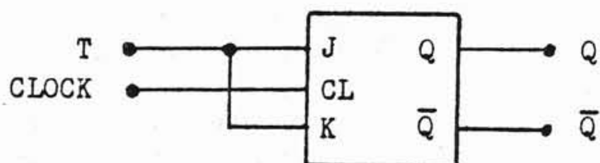


J	K	Q
0	0	Qn
0	1	0
1	0	1
1	1	$\overline{Q}$n

WHERE:
Qn = value of Q during previous clock cycle.

V. TOGGLE OR T-TYPE FLIP-FLOP:

The T-Type F. F. is used primarily in counters. It can be effectively made from a J-K (Master-Slave) F. F. by:



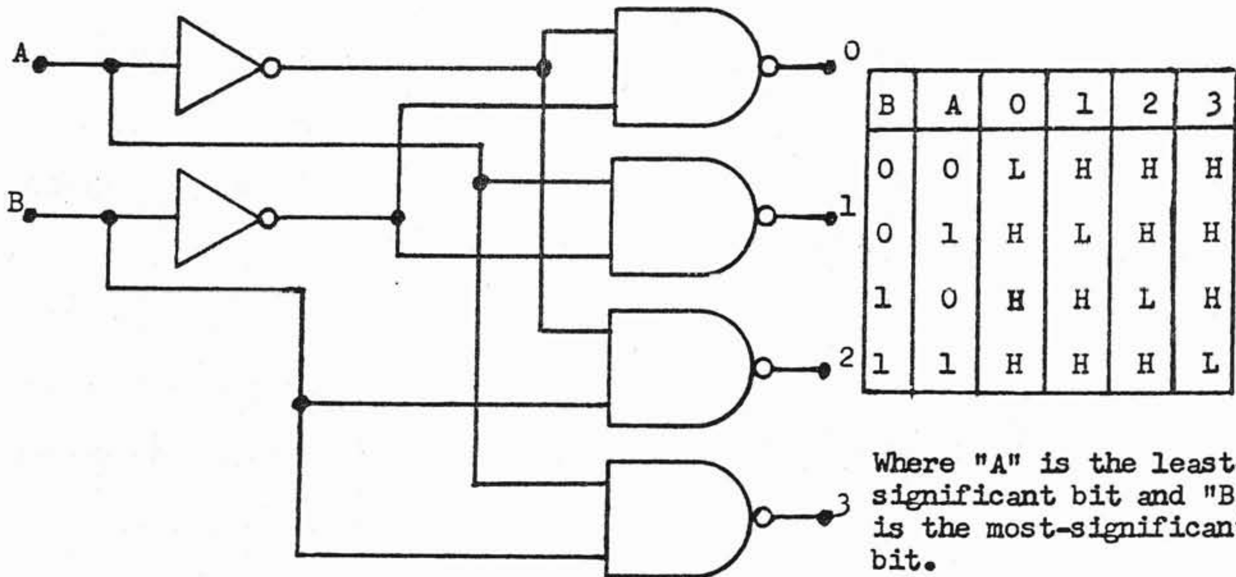
CL	Q	$\overline{Q}$
1	1	0
0	1	0
1	0	1
0	0	1

For every complete clock cycle (⏋⎿), Q and $\overline{Q}$ go through ½ of their cycle. Therefore, the T-Type F. F. divides the clock frequency by 2 as long as the "T" input is held high.

DECODERS/DEMULTIPLEXERS

DECODERS:

These devices are most commonly used for address decoding. They are available in 2-line to 4-line, 3-line to 8-line, 4-line to 10-line, 4-line to 16-line configuration. For simplicity, a 2-line to 4-line decoder is shown below:



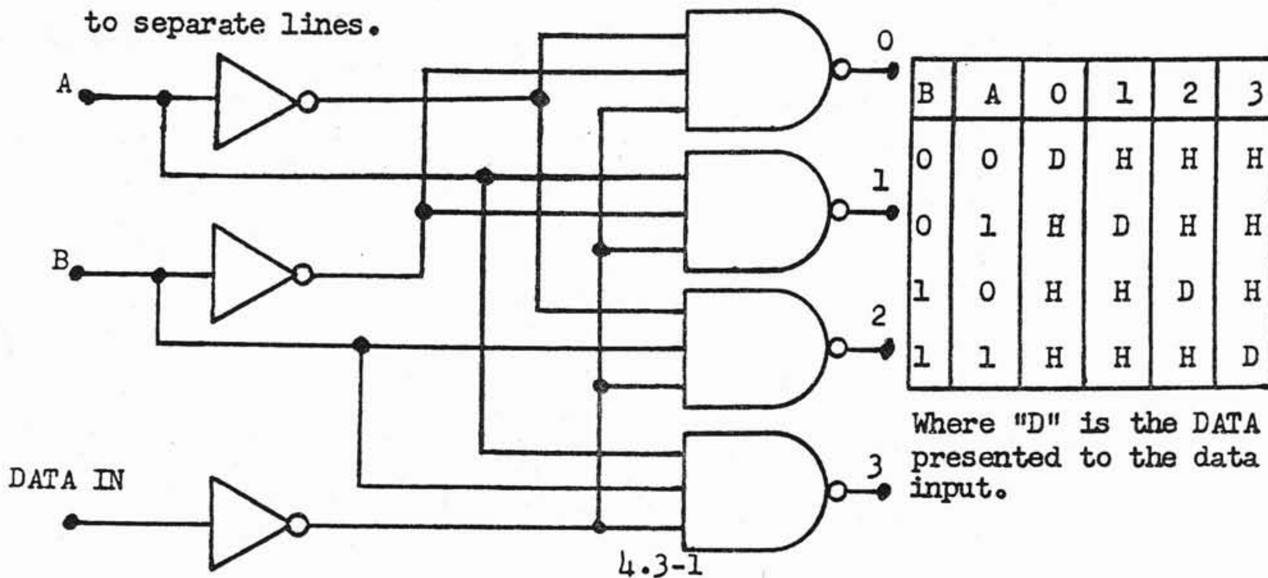
B	A	0	1	2	3
0	0	L	H	H	H
0	1	H	L	H	H
1	0	H	H	L	H
1	1	H	H	H	L

Where "A" is the least-significant bit and "B" is the most-significant bit.

With this device, it only takes 2 lines to specify or enable 4 different devices. The output is low-true.

DEMULTIPLEXERS:

These gates are the same as a decoder, except the NAND gates have an additional input for data. This device separates serial data on one line to separate lines.



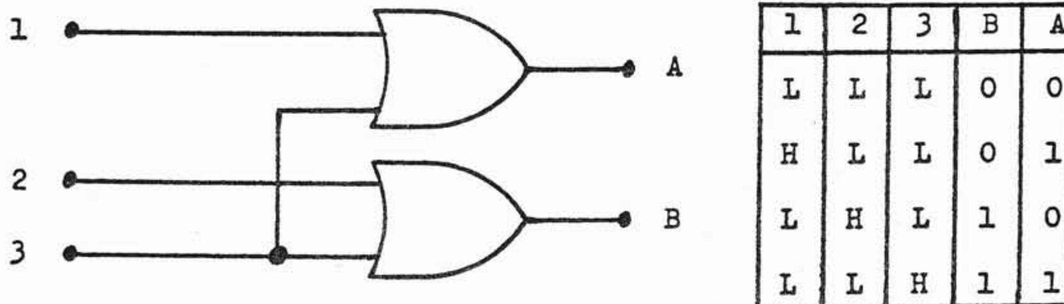
B	A	0	1	2	3
0	0	D	H	H	H
0	1	H	D	H	H
1	0	H	H	D	H
1	1	H	H	H	D

Where "D" is the DATA presented to the data input.

ENCODER/MULTIPLEXERS

ENCODERS:

These devices are used to convert several inputs into a few encoded lines. These are used on keyboards and multi-position switches.



1	2	3	B	A
L	L	L	0	0
H	L	L	0	1
L	H	L	1	0
L	L	H	1	1

MULTIPLEXERS:

Multiplexers or Data Selectors are used to select one of several data sources and place the data from that source onto a single output line. These are available in 4 to 1, 8 to 1, and 16 to 1 configurations.

B	A	CØ	C1	C2	C3	Y
0	0	0	X	X	X	0
0	0	1	X	X	X	1
0	1	X	0	X	X	0
0	1	X	1	X	X	1
1	0	X	X	0	X	0
1	0	X	X	1	X	1
1	1	X	X	X	0	0
1	1	X	X	X	1	1

X = DON'T CARE

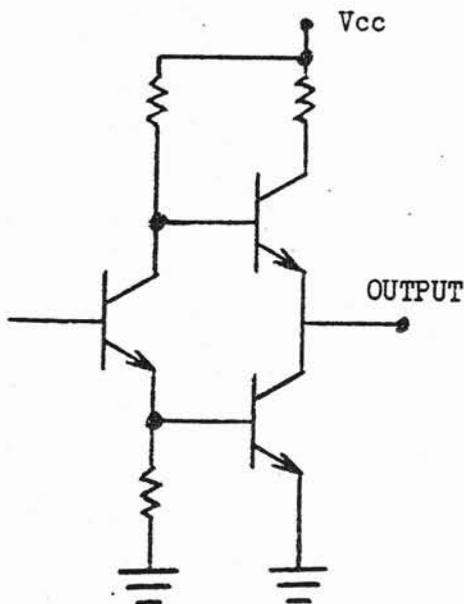
INTERFACE DEVICES

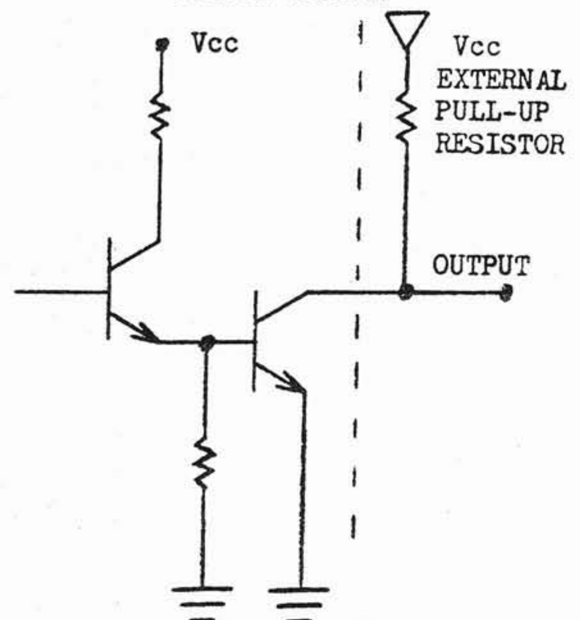
OPEN-COLLECTOR LOGIC

There are several instances where a large multiple input OR gate is needed. In certain cases the common practice is to create a WIRED-OR. This is done by wiring two or more gate outputs together to create a single input into another gate. The WIRED-OR is a LOW-TRUE output. The WIRED-OR is used frequently to OR several interrupts together.

BUT, this procedure CANNOT be done with just any logic gate. The standard TTL logic gate has both active pull-up and active pull-down. Therefore, if two standard TTL outputs were tied together and one output was high, while the other was low; each gate would try to make it's output prevail until finally one of the output transistors of one of the gates burned out.

The only type of logic that can be WIRED-OR together is OPEN-COLLECTOR logic. The internal differences of the output driving circuits is shown below:





NOTE: The open-collector logic has no internal pull-up device (neither active nor passive). Therefore, the gate can ONLY pull what is attached to its output to ground. Since there is no pull-up device in the gate, several of these types of outputs can be wired together with no ill effects. But their combined outputs must have two states (high and low) to be of any use as an input to another gate. Therefore, an external pull-up resistor must be added to the junction of the WIRED-OR. The value of the resistor is calculated by:

$$\frac{2.6}{I_{TL}} > R > \frac{4.6}{I_S - I_{TL}}$$

WHERE:

I_{TL} = Total of the leakage currents of all the gates of the WIRED-OR when their outputs are all high.

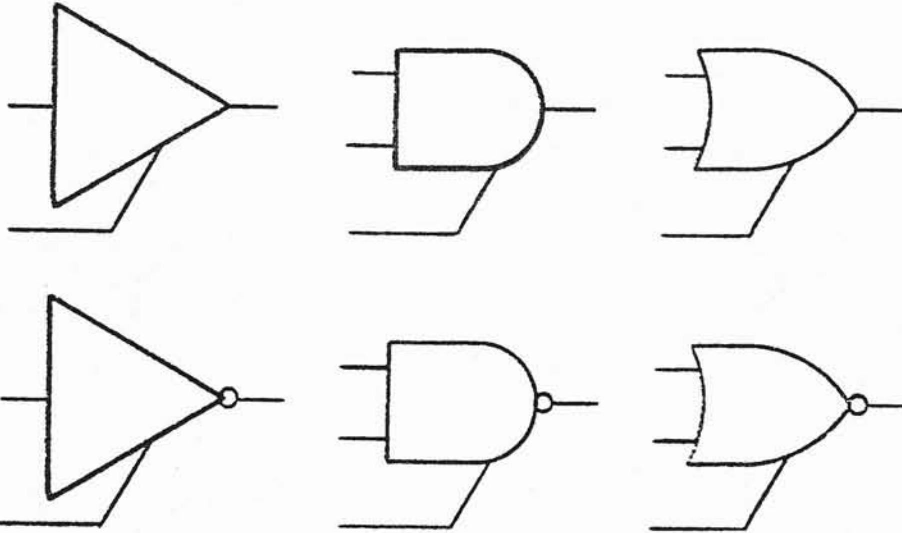
I_S = The lowest maximum current sinking capability of any of the gates forming the WIRED-OR when its output is low.

TRI-STATE LOGIC

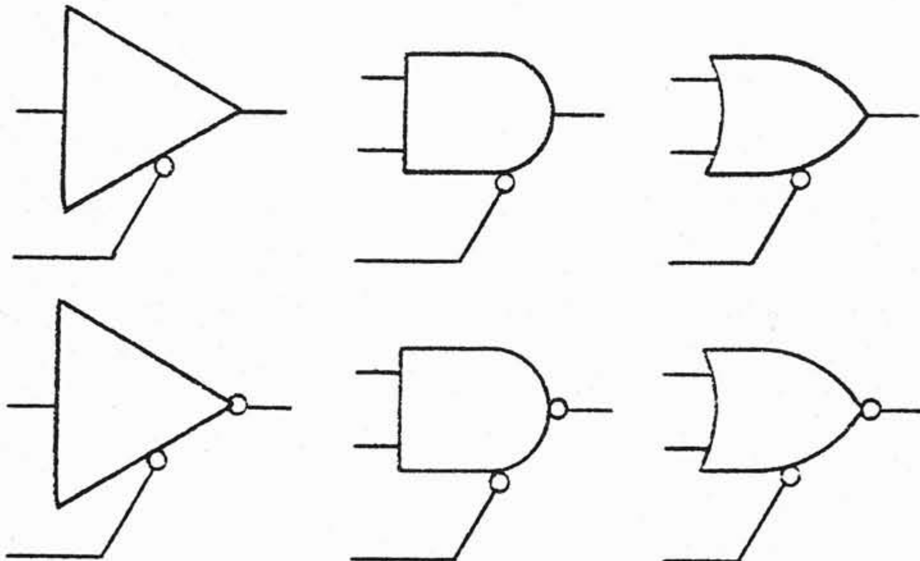
In a micro-computer system transferring of data from one part of the system to another is done via the DATA BUS. In a large number of systems, the number of devices attached to the data bus exceeds the load driving capabilities of the microprocessor or other devices that are connected to it. Therefore, there is a need to buffer the sections of the system to the data bus. There is always more than one section connected to the data bus, so for intellegent communications, one and only one can communicate to the bus at any one time. Therefore, there is a need to turn off or disconnect all but the section that has been enabled by the processor.

But a large number of devices only have two output states (high or low). So, there is a need for a special output that has three states (high, low, or off). This is referenced to as three-state or TRI-STATE logic. The logic symbols for these devices are below:

HIGH-TRUE ENABLES



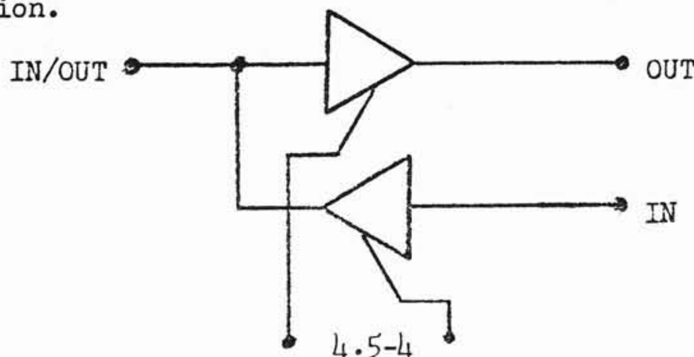
LOW-TRUE ENABLES



These gates when enabled, through the separate enable input, will function like the standard gates that we have already discussed. But, when they are disabled their outputs go to a high-impedance or off-state. Therefore, many 3-state devices can be attached to a common line without unwanted interaction as long as one and only one is enabled to output to that line at any given time.

BUS TRANSCEIVERS

The TRI-STATE devices that have been discussed are essential to one-way communications to a bus, BUT, the processor and a number of other devices are by-directional and need to communicate in both directions with the by-directional data bus. This caused the creation of BUS TRANSCEIVERS. BUS TRANSCEIVERS are effectively two TRI-STATE buffers strapped together in such a manner as to tie the input of each buffer to the output of the other. One of the junctions is to be attached to one of the data bus lines while the other junction is attached to the same respective line of a device or section that is to be buffered. One and only one gate is enabled at any given time. The gate that is enabled is determined by the desired direction of communications (IN or OUT). This is usually done by the READ/WRITE control line. There are usually four strapped pairs in one IC. In some Bus Transceivers, one of the junctions (input to output) is not made within the IC to facilitate interfacing a bi-directional bus to a split data bus or device. If this is not desired or needed, the user can externally make the connection.



ANALYZING SOFTWARE PROBLEMS

ANALYZING SOFTWARE PROBLEMS

INTRODUCTION

The object of this chapter is to present a general procedure used to design software to solve a problem. This procedure is completely machine independent, and it can be applied to any software problems you are likely to encounter. The most important thing to remember about this procedure is that you do not concern yourself with the programming language details until well into the solution. This is true of even the seemingly "trivial" programs. There is no way more certain to result in a program that is sloppy, ill-designed, and hard to debug than to try to write the program directly from the problem definition. To be effective software must be designed first and then implemented using the correct techniques.

5.0 The Software Design Procedure

The systematic approach to developing a programmed system is a logical extension of the normal problem solving cycle engineers and scientists have employed for years. It consists of seven basic steps:

1. problem definition,
2. problem partitioning,
3. algorithm development for each partition,
4. writing the program for each partition,
5. debugging each program,
6. integrating the programs back into the system, and
7. final system debug.

Using this technique, the problem is broken down into smaller and smaller sub-problems until they are a size which you can deal with conveniently and effectively. This is because it is much easier to focus your attention on one small section of the system at a time. You develop each of these blocks and sub-blocks into a group of detailed flowcharts and programs, each of which is tested and debugged. They are then interfaced and the whole system tested. This systematic approach is intended to help you minimize errors, since the small highly localized programs are much easier to thoroughly check out than a single large, spread out program.

Graphically, the procedure is illustrated in Figure 5.1. You start with a central problem and partition it into logical blocks, solve and debug each of the blocks, and finally integrate and refine the blocks into the final system. There may be one or many levels of blocking, depending on the complexity of the problem. With experience, you will find this general approach to be the most direct and consistent way to implement a working software system, regardless of size. Less organized approaches may work for smaller systems, but you will become hopelessly tangled as the systems grow in size. It is best to learn the general procedure and use it on all problems, small or large. The greatest disasters usually occur when the whole design procedure is dispensed with because the problem is too "trivial" to warrant the general approach. Conversely, dogged application of this approach can make many formidable problems turn out far better and faster than anticipated.

In the remainder of this lesson we will initiate our study of the general software solution procedure. Lessons Three through Ten will then expand and refine the techniques used during the solution process.

5.1 Step 1: Define the Problem

As with any procedure for solving any problem, the first step is always the same (and the hardest): define the problem. For the case of software problems, you must decide exactly what the finished software system is to do. This definition of the operational characteristics you want the final system to have is called the functional specification. Naturally, it is easier to define and specify solutions for some type of problems than for others. Problems which are concerned with the implementation of specific features are generally easiest. Problems which require both judgement and implementation are the hardest. In the first case, the task is to figure out how to do something. In the later case, it is often a question of whether or not the job can be done, and if it can, what is the best way to do it. For example, a program to write single data bytes onto a magnetic tape unit is a fairly specific problem with a similarly straightforward functional specification. There is little conceptual design work to be done. It is mainly a question of using a program to control the

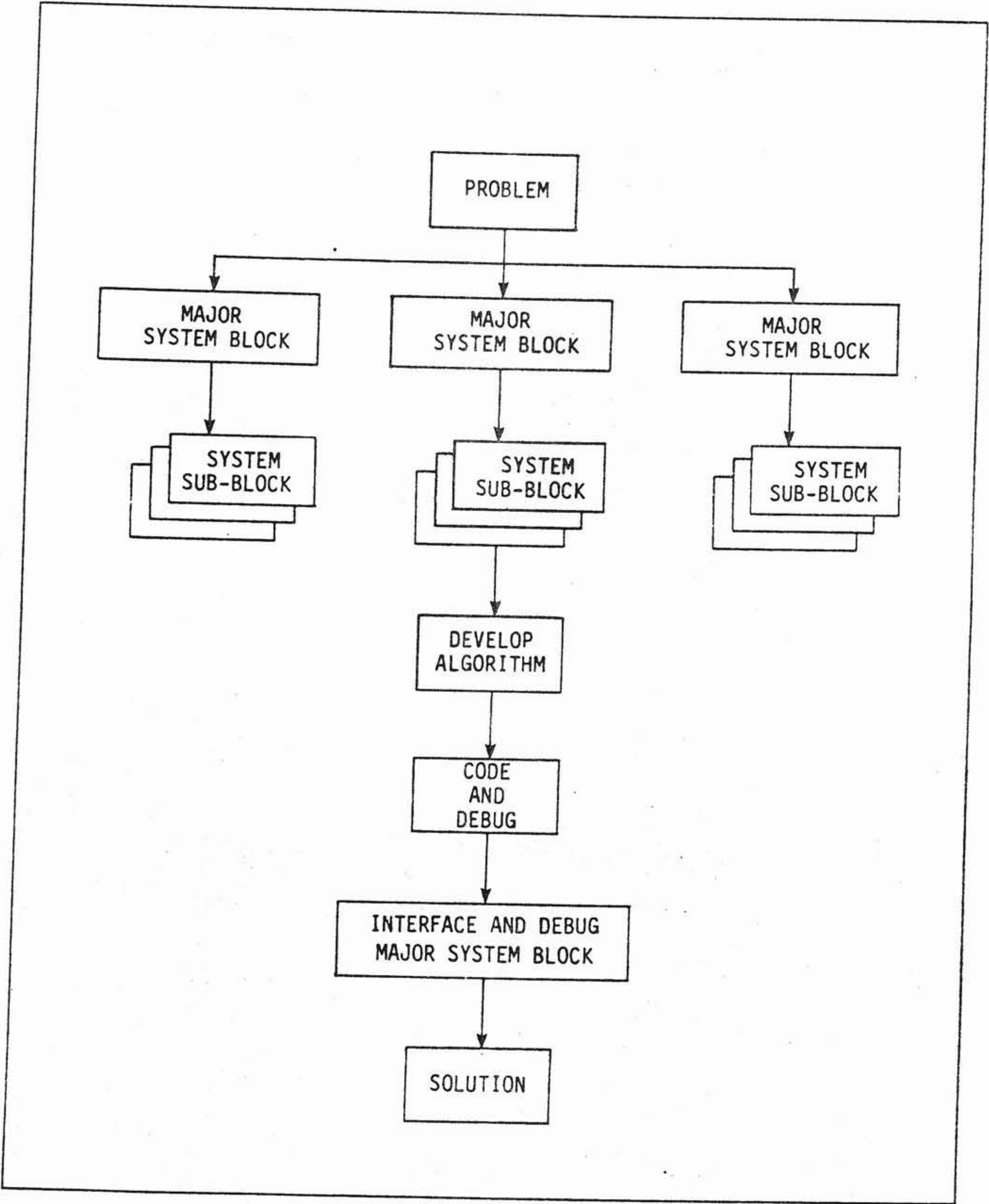


FIGURE 5.1 GENERAL PROBLEM SOLUTION PROCESS

selected hardware. On the other hand, the program required to use this program as part of a system to format sequences of data bytes into records on the tape will require considerably more design. You will have to decide on record length, record marks, whether or not you want to format the data with parity and/or check sum, and so on. Not only that, you must decide on the probable usage of the routine. The quick format program required to test a tape deck's operation in the lab is apt to be quite different from a general usage exchange format for a tape library. In the second case you must consider problems of compatibility with different hardware, reliability, user documentation, and many other details. All of these questions should be settled in the functional specification before you proceed to the next design phase. We will examine both of these cases as examples of general problem solutions in this and later lessons.

5.1.1 Information Required For A Functional Specification

It is difficult to give a complete definition of information that is always required for a functional specification. It varies widely from problem to problem. Simple systems can be specified adequately in a few pages. Large, complex systems may have hundreds of pages of specifications and still be inadequately defined. However, the following information should always be present.

1. A concise problem statement. One short paragraph describing the problem the system is being designed to solve.

2. Required hardware. You must know what signals and devices are available or required. The exact I/O or memory addresses are not important at this point, but you must know the hardware you will be using.

3. Required software interfaces. When designing programs, you will often be placing them into systems where they will have to co-exist with or utilize other programs. If this is to be the case, it should be noted in the specification. In this case, exact details are necessary; you should mention the

relevant system standard or format (i.e., all output must conform to system I/O standard 1-13) for all routines to be interfaced. These requirements will often have a significant affect on your design.

4. A complete description of how the system is supposed to function when complete. This is usually the longest part of the functional specification. This section should include a description of user interaction (if any), data required, output produced, special features, error condition handling, etc. In other words, a complete description of how the system will look to the world from the outside with no consideration for how it will look from the inside.

This problem makes writing the software specification sound like a rather formidable task. It is. A good, well thought out specification is the key to a good (i.e. successful!) project. It is well worth the time required to think the problem through completely. If you know what you have to do, it becomes much easier to proceed directly to a solution than when you must constantly stop and start to fill in the blanks in the problem definition. Few specifications are ever totally complete, but you should strive to get as close as possible before you start the actual design. Once you become immersed in the details of the solution, it becomes much more difficult to separate the normal implementation problems from those caused by a fundamental design logic error.

Example 5.1

Consider the design of a program to interface a magnetic tape recorder to microcomputer. This program will control the transfer of parallel data between the tape deck and the microcomputer. It will control all tape deck hardware functions which are required to perform these data transfers. The following is a possible functional specification.

Scope: This specification covers the program required to interface a Magbyte Model 1010 digital magnetic tape drive to an everyday microcomputer.

Required Hardware: The interface will require the tape drive to be connected to the computer through two input ports and two output ports: one data input port with parallel data from the tape deck, one data output port with parallel data out to the tape deck, one status input port and one control output port. Status input signals available are End of Tape, Write Protect and Ready. Control signals required are Tape Advance, Read/Write and Transfer Data. The timing waveforms are shown in Figure 5.2.

Software Requirements: The I/O routines must conform to the normal system requirements: output data to be passed via the C register (or the appropriate register or memory location for your system) and input data is to be returned in the A register (or the appropriate register or memory location for your system) upon exit. The routines must restore any registers or memory locations used.

Operational Description:

Input Operation: Upon call, the routine will generate all timing and control signals required to transfer one data byte from the tape in the tape drive into the processor. It will then return to the calling program with that data byte. If the tape drive status indicates End of Tape, an error indicator should be set on return. Otherwise it should be reset.

Output Operation: Upon call, the routine will generate all timing and control signals required to transfer the data byte passed from the calling program onto the magnetic tape in the tape drive. If the tape drive status indicates End of Tape or Write Protect, an error indicator should be set on return. Otherwise it should be reset.

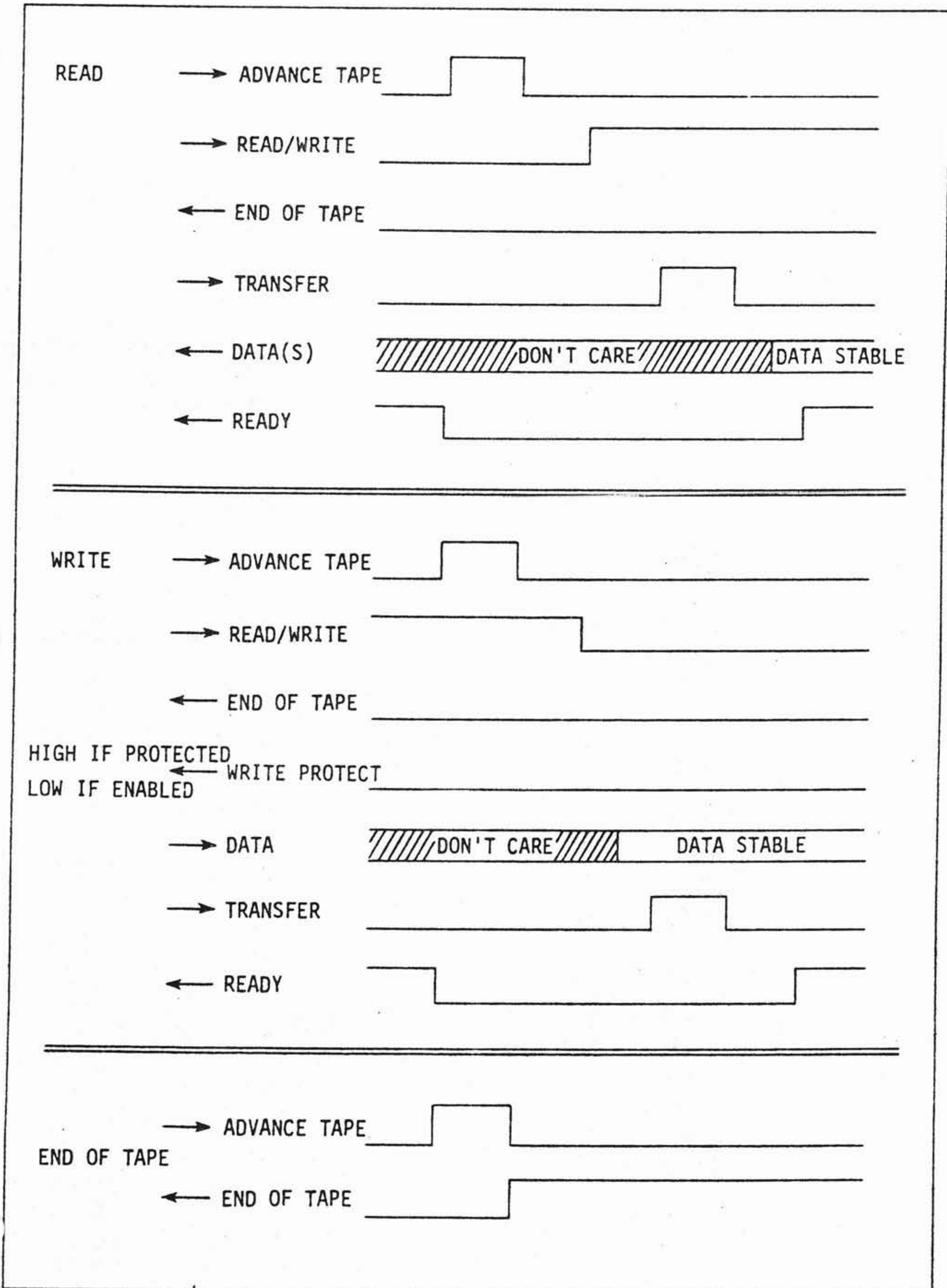


FIGURE 5.2 TAPE DECK TIMING WAVEFORMS

The above example is the specification for an I/O driver routine. All an I/O driver does is control the transfer of data between the computer and an I/O device. Note that the specification makes no mention of the requirements for initialization of the tape drive, how the data is to be formatted, etc. This is because an I/O driver is strictly concerned with transferring data to or from the device it interfaces. It is the responsibility of those programs which utilize an I/O driver to interpret the data and signals returned. A complete tape I/O system which will use this driver will be discussed in Lesson 7.

5.1.2 Using the Functional Specification

The functional specification is the base upon which you will build your system. If it has been properly designed, it will support and guide the rest of your problem solving effort. If it has not been properly designed, your project is probably doomed to failure or overrun before you even get started. Therefore, once you have established a functional specification, use it. If you don't, you are apt to run into that dreaded software disease known as "creeping features". This happens when an inadequate or disregarded problem specification allows non-specified "neat" features to creep into the system after work has begun. This can be disastrous, because changes easily accommodated in the planning stage can require massive effort and re-design work during the implementation stage. Usually, the farther work has progressed, the more effort is required to make any significant changes. The disease is often well advanced before detected and it can be fatal to even the best software projects. (Professional engineers note: marketing departments are notorious carriers of this disease. While they seldom show any symptoms, they are known to infect entire departments.)

The above comments should not be construed to mean that advanced features are to be shunned or omitted. Far from it. The microcomputer makes these features both possible and attainable. What is meant is that they should be designed in from the top, not added from the side. Therefore, when you design your functional specifications, take some time. Brainstorm for a while and come up with a list of features that the system can really accomplish. Try trading off some hardware and software to lower cost or increase

system performance. Microcomputers make whole new fields of features possible, and it is worth your time to see if you can find some for your project. But once that functional specification is done, stick with it. If really drastic changes are needed, you will probably be better off starting over than trying to patch an inadequate specification.

5.2 Step 2: Partition The Problem Into Function Blocks

Once you have completed the functional specification for your system, you can begin to partition it into operational blocks. An operational block is a section of the system which is responsible for performing some specific system function. Operational blocks can be as complex as a complete floating point arithmetic package or as simple as a few instruction data conversions. In system operations, control passes from one functional block to another as the program executes. In this respect the block diagram can actually be considered as a type of overall system flowchart. It differs from the flowchart in that it does not specify the actual algorithms used to implement the functions (see Section 5.3). You first design the structure of the program as a series of successively more detailed operational blocks until you reach a level of complexity that you can deal with effectively. You then proceed to algorithm development for each block.

Blocking and partitioning are the cornerstone of converting a functional specification into a functional program. You can have as many levels of blocks and sub-blocks as the problem requires. When you are first learning, you should not hesitate to block down to sections which seem almost trivial. As you gain experience you will be able to judge more accurately what size blocks you can comfortably handle. Also, extremely involved or complex sections of a system may require much more detailed blocking than the more straightforward sections. The flexibility of blocking is that it allows you to easily adjust the level of detail to match the complexity of the problem.

5.2.1 Deciding on the System Blocks

The decision of what blocks to divide the system into initially is usually made by referring to the characteristics defined in the functional specification. Some common initial blocks are:

a. input operations,

b. program functions (transfer data, search memory, do arithmetic, etc.),

c. system control and timing,

d. output operation, and

e. major data structures (tables, lists, etc.).

These blocks are then drawn and interconnected to form the system block diagram. It is important to remember that at this initial point you are concerned with identification of the major system structures. You are not yet concerned with their actual operation. The design of how the operational blocks will implement their functions will commence once the overall system structure has been established. In theory, it should be possible to implement the system in either hardware, software, or some combination of hardware and software at the end of the blocking operation. This leaves you with the maximum flexibility for actual system implementation.

Example 5.2

Let's take the specification for the magnetic tape I/O driver we wrote in Example 5.1 and do the block diagrams for that system. We can see from the functional specification that we will require blocks to input data, output data, and control the data transfers. Figure 5.3a shows an initial block diagram for this simple system. Note that it shows all data and control signals that are passed through the system. Since the data is transferred to and from the tape deck in parallel form, no further blocking is needed for the Input and Output blocks. However, the control block is required to perform several operations. It must detect end of tape, control the read/write line, sense a write protect condition, and advance the tape. This block is sufficiently complex to warrant sub-blocking. It is shown sub-blocked in Figure 5.3b. Note how all inputs and outputs of the sub-block diagram match those on the main block diagram. It is the same interface expanded to show more detail.

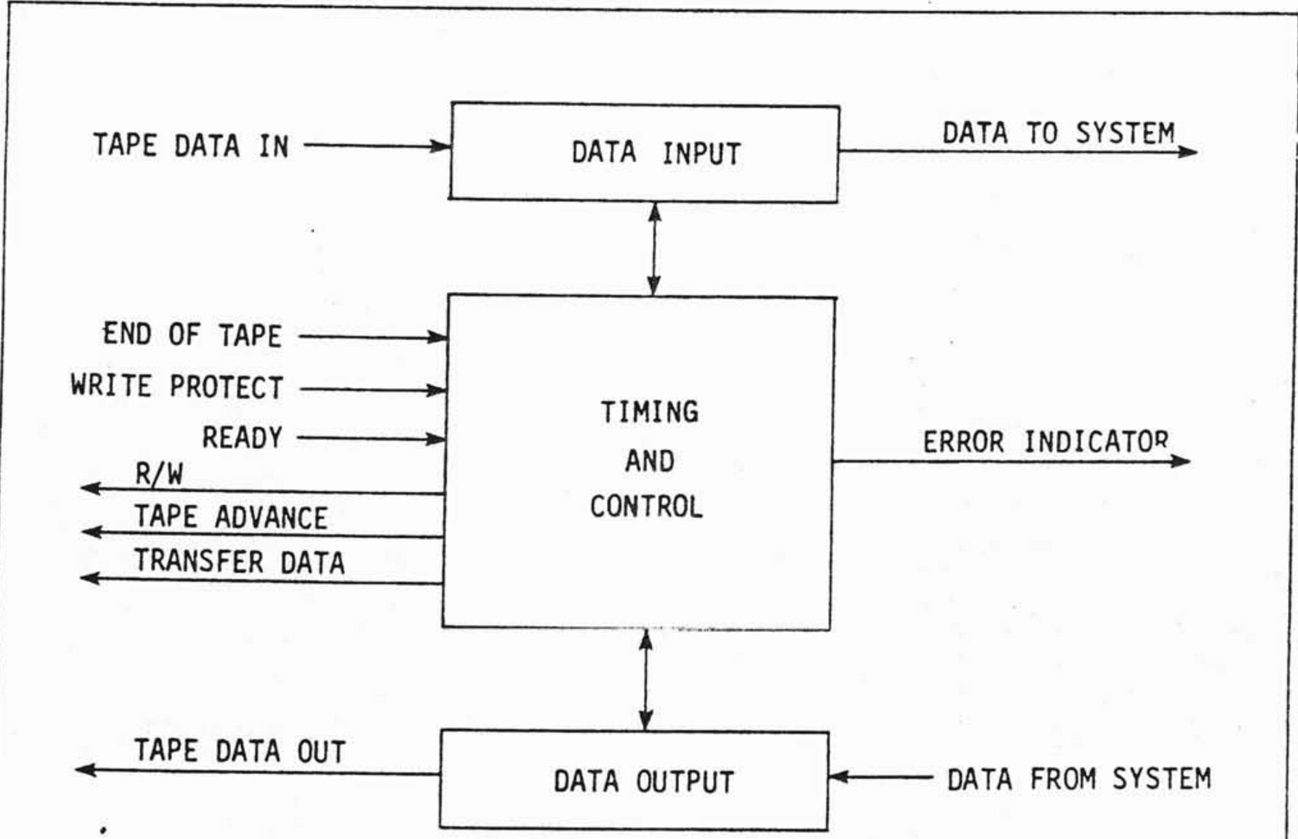


FIGURE 5.3a MAGNETIC TAPE I/O BLOCK DIAGRAM

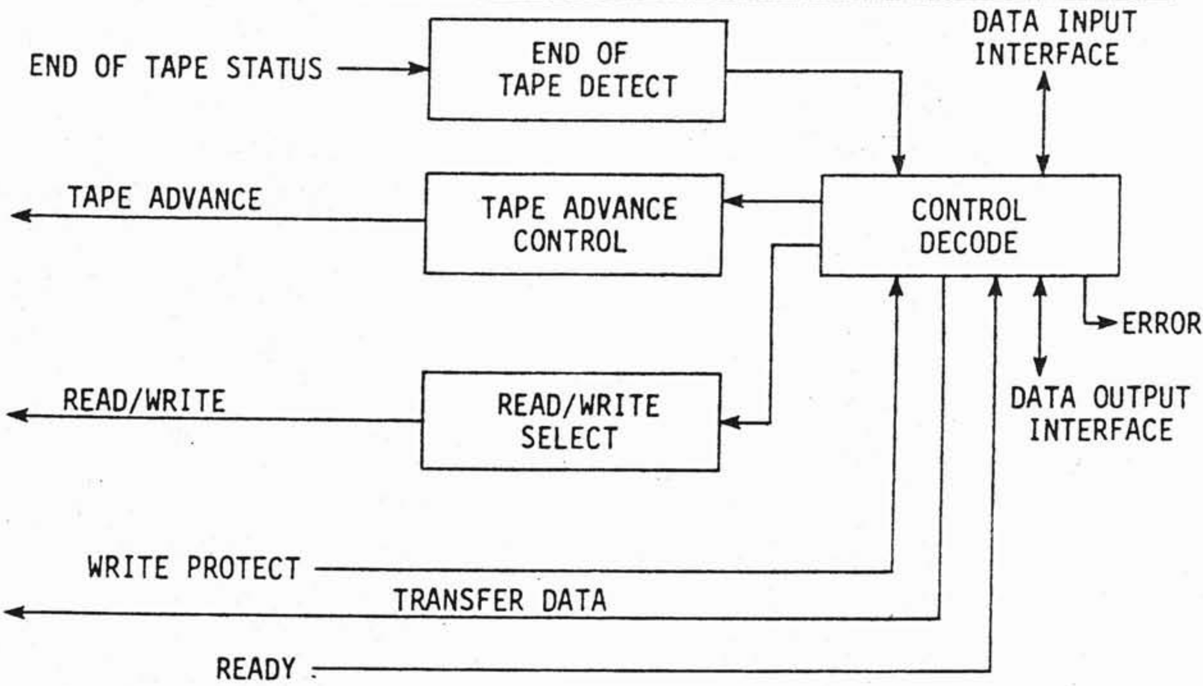


FIGURE 5.3b TIMING AND CONTROL SUB-BLOCK DIAGRAM

FIGURE 5.3

5.2.2 Checking the Block Diagram

Once you have blocked out the system, step back and see if it will meet your functional specification. Be sure you have accounted for all inputs, outputs, data transformations, systems functions, error conditions, and so on. A useful test is to list all the required system features and verify that you have included all the blocks required to perform these functions. After you have confirmed that everything is there, be certain that the blocks are detailed enough for you to proceed on to the logic design implementation. If some of the blocks sound vague or only partly defined, you may need to add more sub-blocks in that area. Repeat this procedure until you are convinced the system defined by the block diagram matches your functional specification. Once you are satisfied that you have covered all the required functions in sufficient detail, you are ready to proceed to the next step and begin designing the actual logic functions required to implement the system blocks.

At this point it is important to recognize that while we are going to continue using the assumption that we are designing a software system, this is not always the case. The problem specification and blocking methods we have presented so far are perfectly general; they can be applied with equal facility to hardware, software, and hardware/software system designs. In the latter case, the optimum trade off between the two implementation techniques will be looked for at this point. Background Section C is devoted to how these fundamental design decisions are made.

5.3 Step 3: Algorithm Development For Each Partition

Up to this point we have only been concerned with the functions to be performed on a block (or non-functional) level. With algorithm development we make the transition from logical system partitions to the actual logic required to implement the system. Most of our algorithm development will be done using flowcharts. The flowchart is often mentioned as the most important step in the software problem solution. This is plainly not true. The flowchart is simply a tool in the continuing design process which began with the problem specification. It is no more correct to sit down and start drawing flowcharts than it is to sit down and start writing

machine code. Both operations have their place in the problem solution procedure. Neither is satisfactory alone. Flowcharts are one possible way to conveniently develop and check the logic of the problem blocks for correct operation. Using flowcharts it is possible to develop program logic independent of any specific computer. It is also much easier to find logic errors in the symbolic flowchart than to try and hunt them down once they are committed to program implementation. (This is particularly true with the relatively primitive debug facilities currently provided by microprocessor manufacturers.)

5.3.1 Flowchart Symbols

The data processing industry has a standard set of flowchart symbols and you should adhere to these in the interest of making your work usable to others. (IBM produces an excellent template of all the standard symbols; it is widely available in stationery supply houses.) The most commonly used symbols and their functions are shown in Figure 5.4 (see page 5-14). These symbols should prove adequate for the construction of any flowcharts you will require.

5.3.2 Type of Flowcharts

Flowcharts can be drawn to represent algorithms at any desired level of complexity. The two most commonly used types of flowchart are the <u>logic</u> flowchart and the <u>machine dependent</u> flowchart. A logic flowchart represents algorithm logic in general operating terms with no reference to specific machine features (registers, memory, flags, etc.). The machine dependent flowchart presents algorithm logic within the context of the features provided by some specific processor. It is advantageous to initially draw a logic flowchart for each functional block in the block diagram. These are then thoroughly debugged and used as the basis for the machine dependent flowcharts required for the computer you are using.

If you program in higher level languages, you will hardly ever use machine dependent flowcharts. The logic flowcharts required to define the algorithm to be used are all that are required. This is because all of the machine dependent details will be handled by the language processor.

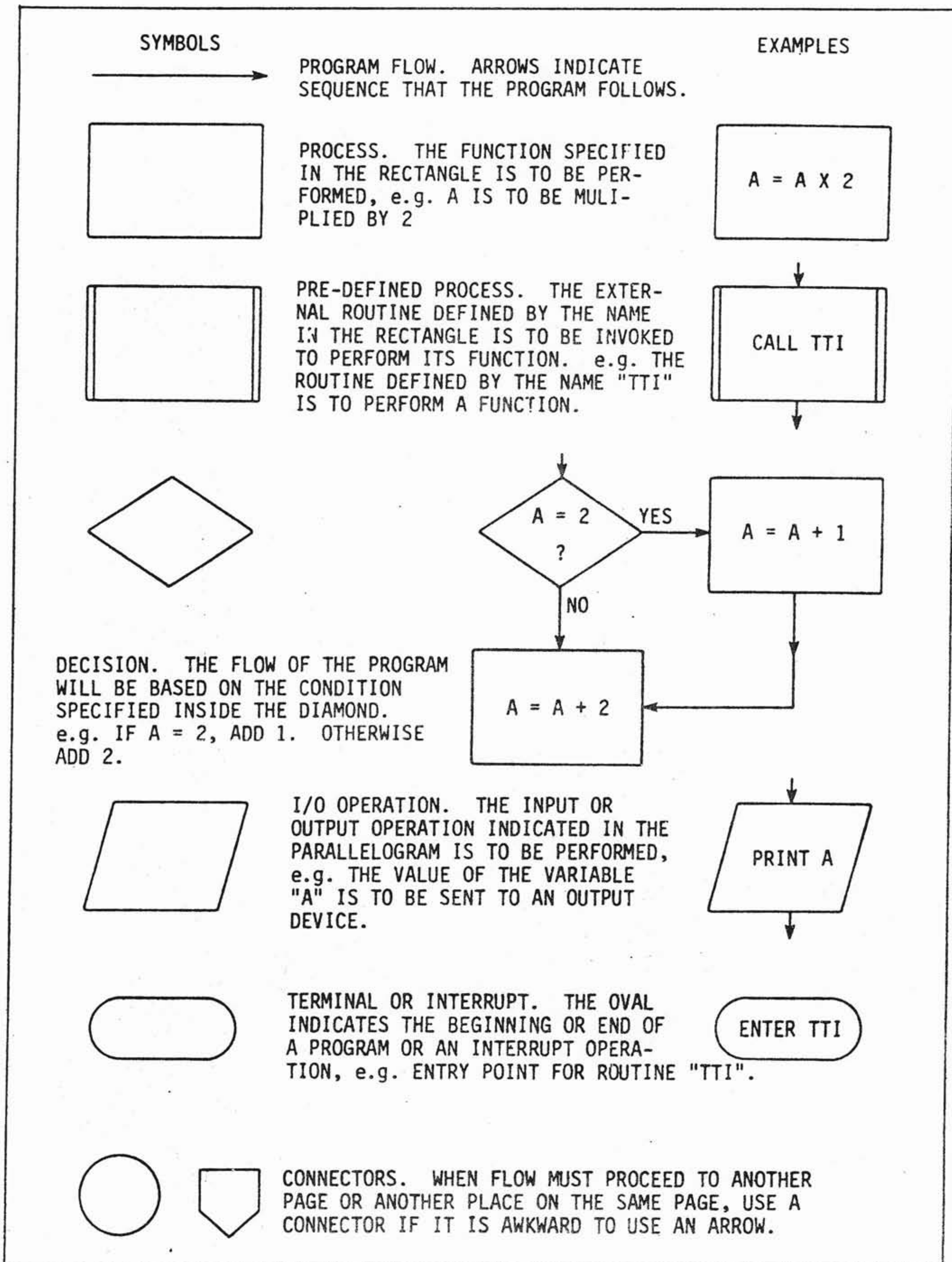


FIGURE 5.4 FLOW CHART SYMBOLS

Similarly, general algorithms and problem solutions which are to be implemented on a variety of computers are best presented using logic flowcharts. Any user can then take the general logic flowchart and use it as the basis for the implementation of a solution on any computer or in any language. As you gain experience with your particular installation, you will be able to go directly from the block diagrams to flow charts that are a cross between purely logical and purely machine dependent flowcharts. However, if you intend to save the algorithm or solution for documentation or possible use on some other system, it would be a good idea to draw a good logic flowchart after the system is completed.

5.3.3 How to Design Algorithms

The design of program algorithms is actually the design of software, a vast subject indeed. We will be covering a portion of that subject in the next eight lessons. However, we can discuss some of the general procedures used when translating a logical system block to an algorithm.

1. Decide what the block is to do. This is the same step as when we initially specified the problem. The only difference is that it is now being done for a small, local program rather than for the whole system. Naturally, the label on the block will provide a good starting place for this description. Usually a one or two sentence description of the operation to be performed is all that is required.

2. Decide where the data to be operated upon is located. Is it read in, passed from another block, looked up in a table, or what? You will need operation blocks to input the required data. While you decide where to get the data, decide if you need to do anything special to it before you use it. Does it have to be complemented? Rotated? Masked? Scaled? If so, you know you will need some data transformation blocks in the flowchart.

3. Figure out how to perform the required operation. This is the real meat of the algorithm development. This will be where you combine process blocks, data and decisions to convert the data from the input

format to the output format. This part of the process will usually account for the largest portion of your work. Remember, developing the algorithm is an iterative process.

It will usually take several tries before you get the algorithm correct. Start out by writing down the sequence of operations to be performed in the order they must be performed, like "read in data, then test for control characters, then test for lower case characters", and so on. This will give you that all important feel for the sequences of actions which are to be performed. After you have the general flow, add the process and decision blocks you need to actually perform the operations.

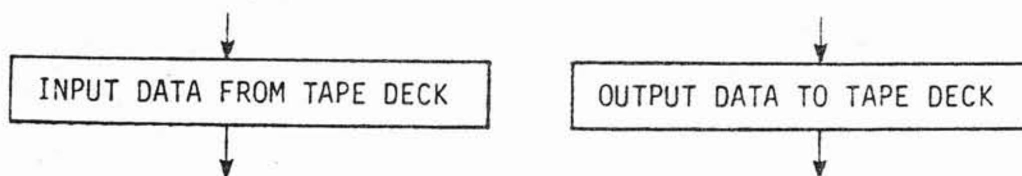
After you have an algorithm that should work, try it out with data to see if it does work (all on paper, of course). Try to imagine every possible data condition that could occur and then be sure your algorithm can process it correctly. You must be certain your logic is correct in the algorithm before you proceed to coding. Be particularly careful that your algorithm can handle error conditions. This is an area which is particularly susceptible to errors which will be very hard to detect. Be patient and thorough. Time spent getting the logic correct in the algorithm will be time saved during system debugging. Think first, program later.

4. Decide what to do with the finished data. Does it have to be specially formatted? Do you save it? Pass it back to a calling routine? Output it? Add the blocks required to get the output data ready for the receiving routine or device.

5. Keep the structure simple. Make it a goal to keep the flow straightforward, logical and clear. Be particularly careful about how you enter and exit from the routines. There are really only a few simple structures you should ever need to use in construction of any algorithm. We will examine these structures in the next few lessons.

Example 5.3

Let's develop the algorithms required for our magnetic tape drive interface system used in Examples 5.1 and 5.2. The first thing that becomes apparent is that the data input and output blocks are very large blocks and very small programs. The data is to pass through the routine in parallel without being modified. Thus the flow charts for those blocks would be simply one block each:



The obvious conclusion is that the majority of these flowcharts will be concerned with when to read and write the data, namely the timing and control blocks. Let's take the read block first. From the timing diagram we can see that for this tape deck the sequence of control for reading a data byte from the tape is advance the tape (from the manufacturer's specification we find that it automatically advances in one byte increments), test for End of Tape, set the Read/Write line to Read, wait for data ready, read the data, then exit. The algorithm for this function is shown in the logic flowchart in Figure 5.5. Note how the flowchart defines a logical solution to the problem without reference to any specific hardware.

A similar procedure is used to design the algorithm for writing data. For Write operation the timing waveform specifies that we advance the tape, test for End of Tape, test for Write Protect, set the Read/Write line to Write, set up the output data, strobe the data transfer line, wait for Data Ready and exit. This flowchart is shown in Figure 5.6. Using these two logic flowcharts we could then draw the machine dependent flowcharts or proceed directly on to the actual program.

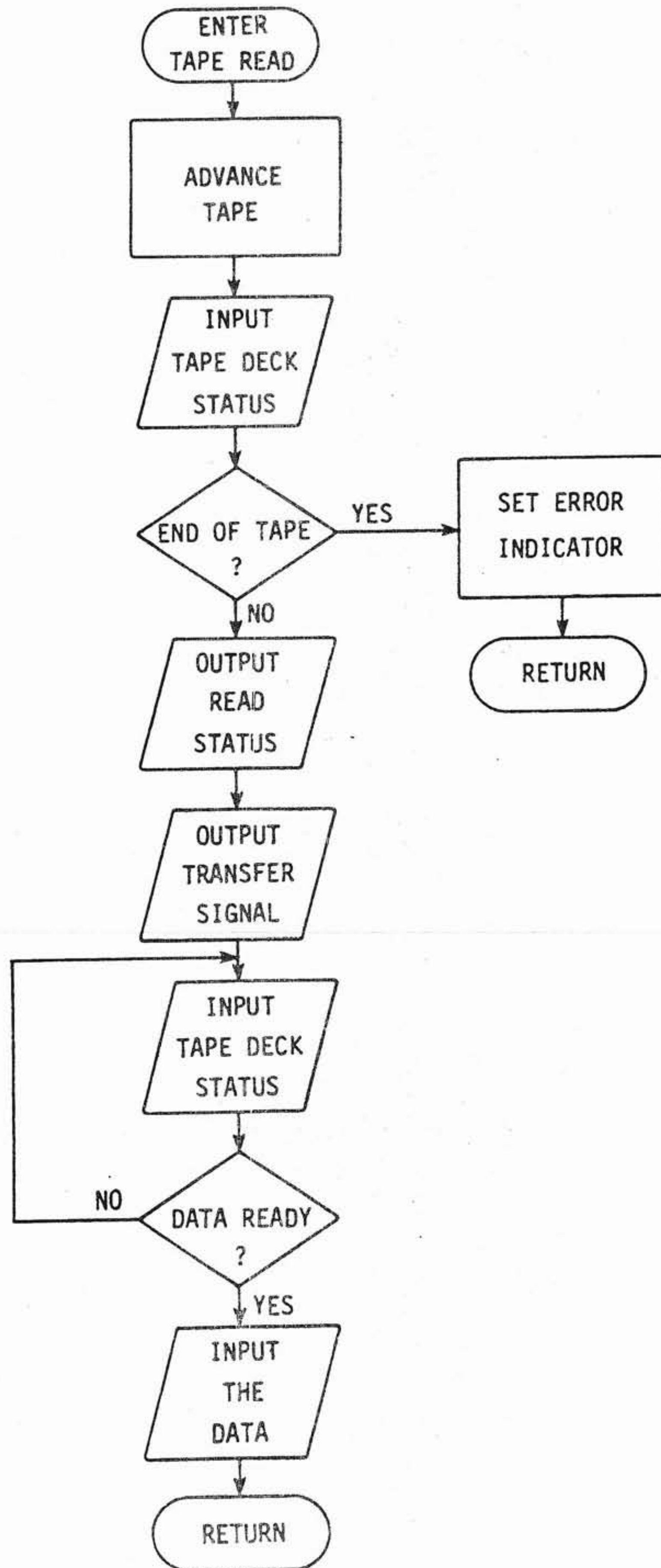


FIGURE 5.5 TAPE DECK READ LOGIC FLOW CHART

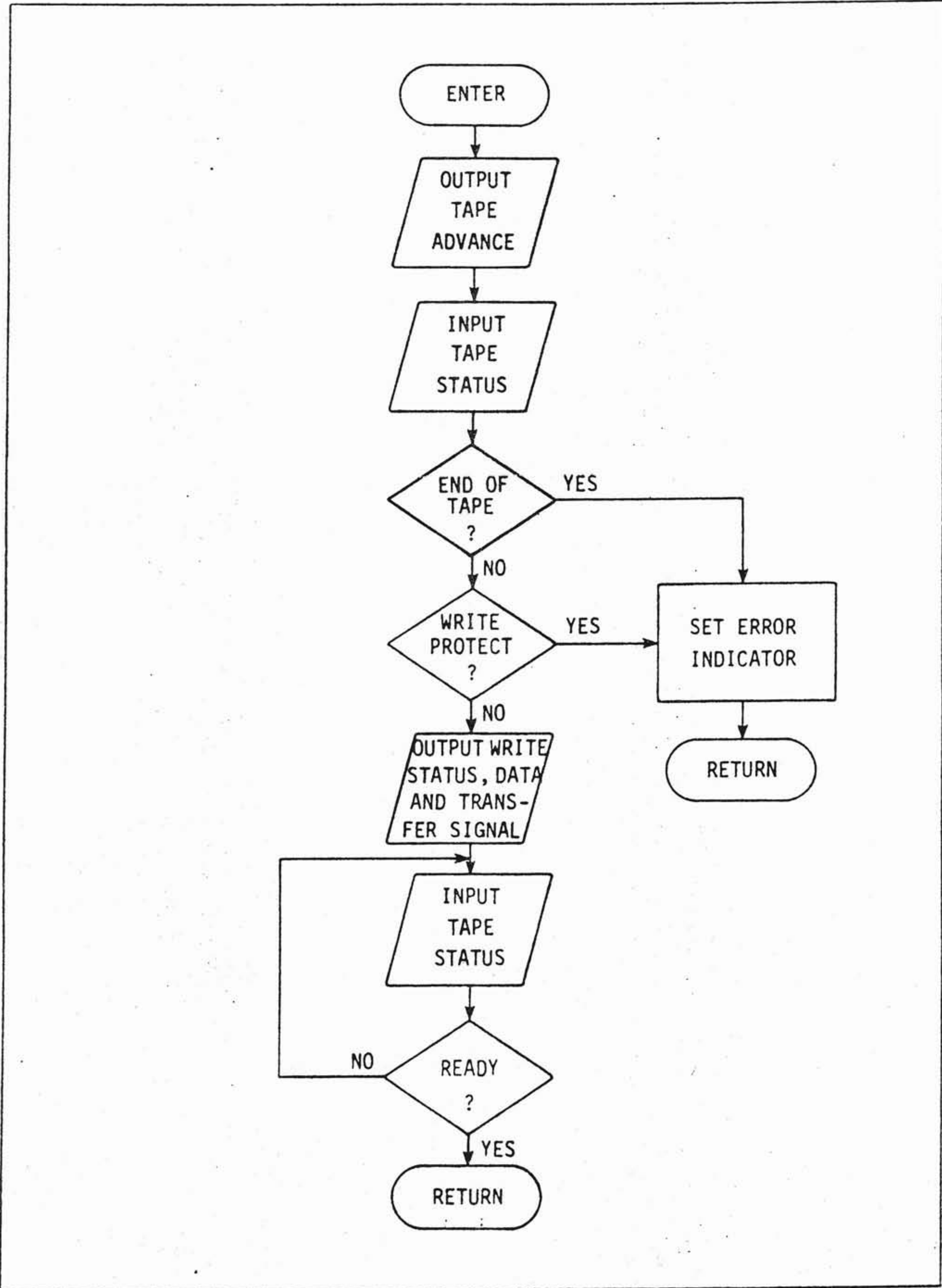


FIGURE 5.6 TAPE WRITE LOGIC FLOW CHART

5.4 Objections to Flowcharts

We have been using (and will continue to use) flowcharts to represent the algorithms we have developed. This procedure is not universally accepted, particularly in the data processing industry. Critics maintain, with a certain amount of justification, that flowcharts are unnecessary and even misleading. This position arises from the basic contention that (1) flowcharts are only marginally useful in higher level language program developments and (2) complex flowcharts can become very difficult to follow. To support this position they cite very valid evidence that most professional programmers draw only very limited flowcharts prior to commencing coding. In fact, most flowcharts for large systems are drawn for documentation purposes after the program is complete. This situation arises from the fact that when writing programs in modern higher level languages, algorithms can be efficiently developed directly in the language with no intermediate flowcharts at all.

To answer these arguments (which we really basically agree with), we must point to two basic facts: (1) satisfactory higher level languages are not yet generally available for microcomputers, and (2) most programmers developing microcomputer programs are not professional programmers. The contention that poorly structured flowcharts are hard to follow is completely true. We will always go to great lengths to keep flowchart logic structure clear.

The first fact, the lack of higher level language availability, is obvious. There are at present only two widely available higher level microcomputer languages (Intel's PL/M* and various BASIC** interpreters.) Of these, only BASIC is available for small system use. It will be some time before common higher level languages such as FORTRAN or COBOL will be available for microcomputers. In the interim, the work will be done in assembly language. Even when higher level language processors become available for microprocessors, the nature of many microprocessor applications is

---

*PL/M is a registered trademark of Intel Corp.**BASIC is a registered trademark of Dartmouth University.

such that a knowledge of assembly language will still be required. Higher level languages are only marginally effective in developing programs for use in control or real time applications. Programs of this type require the complete control of the computer's hardware that assembly language provides. For assembly language, use of the flowchart provides a pseudo higher level language for algorithm development that can be either dependent or independent of the computer to be used. (We will have much more on the higher level-assembly language tradeoffs in Lesson 9.)

That most microcomputer programmers are not professional programmers is also fairly obvious. Most current microcomputer programmers are logic designers and hobbyists, many programming for the first time. Since they will probably be forced to use assembly language, these users will be learning programming, algorithm development, and machine structure all at the same time. The use of assembly language programming and flowcharts will enable us to separate these learning activities. In particular, the initial process of teaching general algorithm development is better presented with general flowcharts than with some specific language. The techniques presented using some specific language may reflect the compromises made by the language rather than those required to solve the problem. After some initial algorithm development training, the user may be able to proceed to flowchart free higher level language programming. For that initial training, however, the logic flowchart is an important teaching tool.

To make maximum use of flowcharts without becoming unduly attached to them we will adopt a carefully structured approach. All algorithms will be presented in general logic flowcharts. We will not use machine dependent flowcharts except in the context of specific examples. All flowchart structures will be chosen from a small group of simple, logically sufficient structures whose use can be directly transferred to most higher level languages. In this way we will make maximum use of flowcharts while avoiding the major objections.

5.5 Procedures After Algorithm Development

After you have completed the problem definition, block diagrams, and algorithms, you can begin to think about writing the program required to implement the logic system you have defined. However, it should be apparent by now that if you have followed the first three steps correctly, this step should present you with very little trouble. The blocking and algorithm steps combined with the flow charts will have supplied the system structure and control logic. All you will need to do is implement these features using the programming language you have available. Naturally, that is easier said than done, but if the logic is correct, the problem will have been reduced to finding combinations of machine instructions or higher level language statements to perform the desired operations. We will spend the next eight lessons refining and expanding your problem solving skills, augmenting these skills with useful programming techniques.

5.6 Summary

This lesson has presented the general approach required to solve software problems. All software problems can be solved by dividing them into blocks and sub-blocks, developing algorithms for those blocks, writing programs to implement the algorithms and interfacing the blocks back into a system which solves the problem. The general approach to problem definition, blocking and algorithm development was then presented and illustrated using the example of a digital Read/Write magnetic tape deck.

QUESTIONS

1. Describe the general software problem solution process. Is this the way you normally approach problems? Do you think the general procedure can be applied to other, non-software problems?

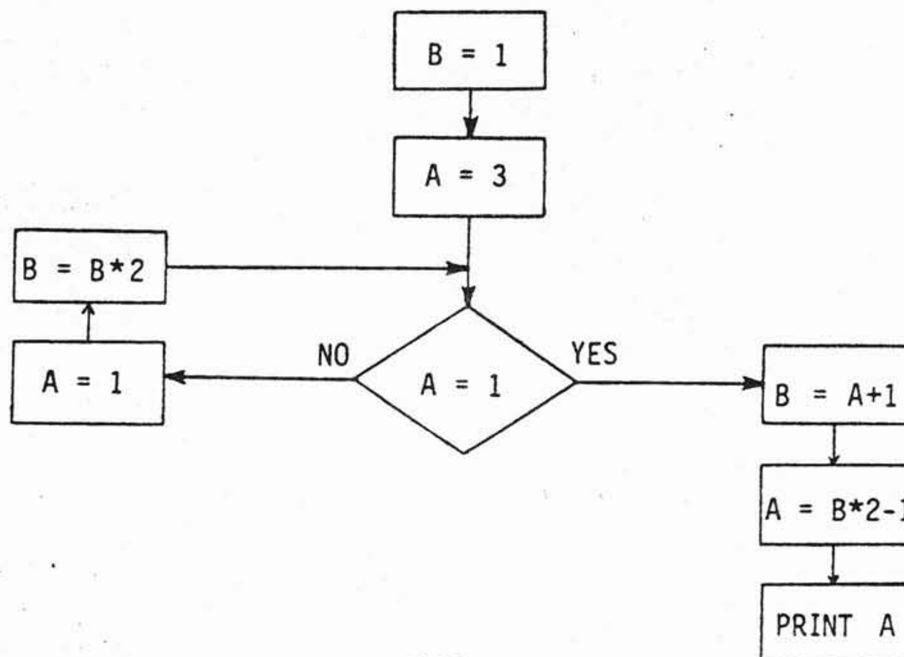
2. Why is it important to establish and follow a functional specification at the outset of the solution to a problem?

3. Describe "creeping features". Have you ever seen it in action? What was the cause? What was the result?

4. Describe the difference between a logic flowchart and a machine dependent flowchart. Which do you usually use? If you usually use a machine dependent flowchart, do you usually draw a logic flowchart of the solution for future use?

PROBLEMS

1. What value of A will be printed in the example flow chart below:



2. The Fibonacci series F(N) is a mathematical number sequence which is defined for all integer values of N by the following algorithm

$$F(0) = 0$$
$$F(1) = 1$$
$$F(N) = F(N - 1) + F(N - 2) \text{ for all } N > 1$$

For example,
$$\begin{aligned} F(2) &= F(2 - 1) + F(2 - 2) \\ &= F(1) + F(0) \\ &= 1 + 0 \\ &= 1. \end{aligned}$$

Thus the Fibonacci series can be represented as follows

N	0	1	2	3	4	5	6	7	. . .	N
F(N)	0	1	1	2	3	5	8	13	. . .	F(N - 1) + F(N - 2)

Draw the flowchart to compute F(N) for any value of N.

3. Draw a flowchart which incorporates the flowchart developed in Problem 2, to compute and print the first 100 values of N and F(N). (Assume that the command "Print" is sufficient to print a value.)

4. One simple method often used to multiply two numbers together is to repeatedly add one number to itself. For example, 3 * 4 can be thought of as 3 + 3 + 3 + 3 = 12. Develop the algorithm to multiply two numbers using this method. Draw the flowchart. Do you feel this is an efficient way to multiply two numbers? Is there any way to make this basic algorithm more efficient?

THE HARDWARE/SOFTWARE APPROACH TO MICROCOMPUTER DESIGN

THE HARDWARE/SOFTWARE APPROACH TO MICROCOMPUTER DESIGN

INTRODUCTION

In the course of designing a system there are a series of crucial decisions which must be made regarding the ultimate system implementation. Throughout the software course we are concerned primarily with the implementation of the software portions of these systems and how they interact with available hardware. To be sure, this area is vital to the designer. However, the thorniest problem initially confronting most designers of microprocessor based systems is how to partition the system functions between hardware and software implementations. This is understandable since most users are far more experienced with hardware design than software design. However, the plain truth is this: within the speed limits imposed by any computer, anything that can be done with hardware can be done with software. In fact, only a small percentage of applications will present speed problems. Usually even these applications are only speed sensitive in areas which can be readily identified and processed with discrete logic to make them adaptable to a software solution. We thus have a sliding scale of implementation possibilities from applications with no software (i.e. no microprocessor) to applications where 95% of the system cost will be in the software. Given this wide range of possibilities, how do we decide where to draw the line? Where indeed. Assuming that the objective is to do the job and make some money, the answer is obvious: we draw the line at the point where we find the lowest cost hardware/software system that does the job.

Before we can discuss how to trade hardware cost for software cost, we must first identify the areas that affect cost in both of these areas. For the purposes of discussion we shall consider cost to be localized in three areas: hardware cost, software cost, and system cost. After we have discussed the various cost areas we will be able to discuss tradeoffs required to modify system cost and performance.

The CPU chosen for the system will have the central effect on the hardware cost of the system. This is not because of the cost of the processor itself. For most systems the actual CPU cost will be an insignificant portion of the total system cost. It is a result of the effect of the CPU on all other aspects of the system design, both hardware and software. It therefore makes the most sense to discuss these costs within the context of the CPU itself.

6.1 Hardware Cost

Hardware cost will be considered to be all of the hardware which must be designed to implement the required system functions. This would include the microprocessor, memories, interfaces, clocks, power supplies, terminals, printers, or other pre-configured peripherals.

6.1.1 System Speed

To paraphrase an old police traffic slogan, "Speed kills microcomputer projects". This is due to the sad fact that of all the great things microprocessors do, doing them fast isn't their best attribute. Most commonly available microprocessors have maximum cycle speeds in the 2MHz range. Execution of an instruction generally requires from 4 to 10 machine cycles. Moreover, doing anything useful will require several instructions. What all this means is that a microprocessor operates considerably slower than conventional sequential and combinational logic. As a rough rule of thumb, if your system requires the processor to do anything faster than 10μsec (100 kHz) you will need to be very careful in your design.

There are a limited number of high speed microprocessors available, but these are sets of devices, not single package microprocessors. They are somewhat harder to use and considerably more expensive. If you begin to use these you may discover your cost rapidly exceeding the cost of some other form of logic implementation. Also, high speed for the CPU generally requires high speed memories, interface logic, and peripheral devices, further raising costs.

As we mentioned earlier, few projects have overall speed requirements that are so severe as to preclude microprocessor use. However, they do exist, and if you think you have one, be very careful to be certain from the start that a microprocessor will be able to do the job. Conversely, there is no point in paying for system speed you don't need. Speed is expensive. You generally get a certain level of speed with the microprocessor. If you're not using it, see if you can trade it for some interface simplicity. No use buying a fast processor and fast interfaces if a fast processor and some slower, dumber, and cheaper interfaces will do. We'll talk more about this later.

6.1.2 Memory Requirements

The system memory is where you will store the programs and data required for system operation. With most microcomputer systems this memory will consist of a combination of read/write memory (RAM) and read only memory (ROM). (With some processors the CPU itself contains a small read/write memory, thus making it possible to implement simple systems with just the CPU and ROM's. Larger systems will require additional read/write memory.) The object of the game here is, as usual, to minimize cost. This is done by getting as much of the software into ROM as possible. This is because ROM can be left with power off and the program will still be there when power is restored. Alas, such is not the case for RAM. Thus when you hear people say that programs should be in ROM because ROM is cheaper then RAM, it isn't really true. Bit for bit the costs are becoming quite comparable, with many types of ROM considerably more expensive than RAM. The fact is that RAM is not practical in dedicated systems which must maintain the program without re-loading memory every time the power is turned ON.

Read/write memory can be broadly divided into static RAM and dynamic RAM. A static RAM will maintain its data as long as power is applied. A dynamic RAM must be "refreshed" periodically. This refresh operation is accomplished by pulsing some of the address lines (usually the most significant bits) periodically. To do this requires the addition of special circuitry to the system. In general, the integrated circuit constraints are such that a static memory requires more area on the semiconductor chip than a dynamic

memory of similar size. Static memories also dissipate more power per bit. The largest RAM memories are, therefore, usually dynamic, at least initially. As the device technology improves these larger memories usually then become available in static form.

The cost of both static and dynamic memories has declined and will continue to decline. This cost is based on the absolute cost per bit for a given amount of storage. However, the device organization and not this absolute cost per bit is often more important in practical applications. In terms of cost per bit, a 4096 x 1 dynamic memory may be much cheaper than a 256 x 8 static memory. However, you will need eight of the dynamic memories to do any good. They will require refresh circuitry, and they will take up eight times more P.C. board space in production. If you only need a 128 byte buffer and some miscellaneous program storage, the bigger "cheaper" memory may cost far more. For cost effective design it is imperative that you avoid memory overkill. Design in what you need, allow some extra for unforeseen difficulties and reasonable future expansion and stop.

The advances in memory technology are impressive and they receive lots of publicity. But the fact remains that few systems for mass production will require vast amounts of RAM. Often minimum package count and ease of system interface will be far more important than sheer volume. Buy one development system with lots of RAM. Use it to develop lots of systems with only the RAM required to do the job.

With ROM's, the situation is considerably different. Read only storage is really only useful organized in multiples of the computer's basic data word. It doesn't make much sense to mask program two 1023 X 4 ROMs for use in an 1024 X 8 system. As a result, ROMs are widely available for eight-bit processors in sizes from 8 x 8 to 2048 x 8. ROMs are available in three types, each suitable for certain areas of application: EROMs, PROMs and masked programmed ROMs.

An EROM is a ROM which can be erased and re-used. An EROM can be programmed and, if errors are found, erased and reprogrammed. Erasure is accomplished

by exposing the EROM to intense ultraviolet light for a half hour or so. In this way the EROMs can be re-used indefinitely. EROMs are the most expensive type of ROM. They are best used in development work or low volume production equipment which require frequent changes to the operating program.

A PROM is a ROM which comes from the manufacturer with all locations as one's or zero's. It can then be programmed by the user. Unlike an EROM, however, once programmed a PROM cannot be erased. PROMs are somewhat lower in cost than EROMs. However, frequent program changes can quickly make them more expensive. They are best used in production systems which will require few changes but whose production volume does not justify a mask programmed ROM.

A mask programmed ROM is fabricated by the manufacturer to contain the desired program. It is neither field programmable nor erasable. A ROM is ordered by sending the semiconductor manufacturer your program. They then generate a custom ROM from your specification. The cost of ROMs produced this way is the lowest available. However, the semiconductor manufacturers charge a flat fee for the generation of the required mask. This cost makes mask programmed ROMs cost effective only for those high volume products whose program will never (hopefully) require change.

6.1.3 I/O Requirements

It is rapidly becoming apparent that I/O is the soft underbelly of most microprocessor based systems. Interfacing the microprocessor to the rest of the system is always a requirement. The microprocessors currently available generally provide only enough interface capability to directly interface one normal TTL device. This means that all signals in and out of the microprocessor must be buffered. Further, control signals must be decoded, interrupts must be processed, data must be latched and held until the processor or peripheral is ready to accept it, and many other system requirements must be met. All this falls within the realm of I/O.

The fundamental element of microprocessor I/O operations is the _I/O port_. An I/O port is the point where the signals to and from the various I/O devices meet their respective signals from the microprocessor. I/O ports provide both buffering and some control decoding. The I/O addresses sent out by the CPU are decoded to provide an enable signal to a specific I/O port, thereby gating the information from that port onto the system data bus for a read operation or gating the information on the system data bus into the port for a write operation. The mechanics of how the port works are not as important as the realization that all data into and out of the microprocessor is going to have to pass through I/O ports. This means that you will want to get your money's worth out of every port. To help you do this, some processors provide a small number (usually two or four) of I/O ports right on the CPU chip itself. If you only need one or two ports for a simple system, this can be a significant cost saving factor.

After you've got the I/O ports, you then must design the special logic required to control the devices or circuits you are interfacing. For most microprocessor applications this is where you will do the majority of your hardware design. If you do lots of microprocessor systems, you will eventually arrive at some standard I/O port design, but there will almost always be some detailed interface design work to be done.

When making the decisions about how to implement your I/O ports and control logic, you may be able to obtain some cost advantage by using a specialized interface device. Some microprocessor manufacturers have designed special families of devices to ease I/O design. These devices usually consist of several I/O ports, some defined logic functions, and all required control logic required to interface some device directly to the microprocessor with little or no external logic. For example, the data ports, control logic, and interface circuitry required to input and output parallel data directly to a serial interface is one popular example. Others include interrupt handlers, real time clocks, bi-directional data ports, and so on, with more becoming available as the industry defines what functions are commonly useful. If you can find some of these to fit your needs, they can save you money.

I/O design is the area where you can often achieve significant savings by trading hardware for software. It is also the area where you may be able to trade some cost for enough added speed to make a usable system. I/O design is an area where creative use of software and hardware will result in optimum system performance at lowest system cost.

6.1.4 Peripheral Devices

In terms of production cost the most expensive portions of your system can easily turn out to be those assemblies you have to buy pre-assembled. All types of computer keyboards, displays, printers, tape equipment, A/D and D/A converters, and similar peripherals are very expensive relative to the cost of the microprocessor hardware. In the normal microprocessor system these devices account for over 50% of the hardware cost. If you must include these components in your system, it is very important to make a very careful analysis of whether or not your product is still cost effective. It can be devastating to have to add a $75 keyboard to a microprocessor system where the total component cost is only $50. In this type of situation you might see if you can use a less expensive device and add the other features with software. All these types of decisions must be weighed carefully before you start the actual design.

6.1.5 Device Support

Into this category we toss all those microprocessor system details that drive your system cost up. These are particularly obnoxious because they are often overlooked until it is too late. The three most common offenders in this category are clocks, power supplies and interface requirements.

The system clock is used to provide the timing signals required to run the CPU and some of the other system logic. From a cost standpoint, there are two areas of interest: who generates the clock and how good does it have to be. In the first case the answer is either the CPU or the system. If the CPU generates its own clock (it may need an external resistor and capacitor), you don't have to worry about the second question. If you have to generate the clock, you definitely have to worry about it. Some microprocessors are very finicky about their clocks. This means special driver

chips, crystals, logic, power supplies (i.e. money). If you are in a very cost sensitive operation, this can make a significant difference.

In addition to the main CPU clock, certain interfaces will require their own clocks. This includes serial interfaces, real time clocks, and many special interfaces. In some cases you may be able to derive the required clock(s) from the main system clock. If not, you will have to plan on the added cost of the required additional clock(s).

Power supplies are another area where requirements differ widely from microprocessor to microprocessor. Some microprocessors will run off the same +5V power supply that is used for all the logic. Some require up to three different power supplies. Power supplies are not cheap and you can quickly add a large cost to the system that you may be able to avoid entirely by chosing a different processor. (Note: after you go to the trouble of picking a microprocessor, be sure the rest of the system runs on the same voltages. It doesn't make much sense to cut corners to get a single supply microprocessor and discover the memories chosen need three supplies anyway.)

Besides paying attention to the number of system power supplies, you must be aware of the overall system current requirements. These requirements can vary widely, depending upon the CPU, memories, and interface logic used. You must be certain that your power supplies can supply enough current to meet peak system usage. Conversely, you don't want to pay for more capability than you need. To solve this problem, you usually don't settle on the final production power supply ratings until the system is complete and its power requirements are characterized. This is in contrast to the selection of the system hardware, where the number of supplies to be used in the system is determined before beginning the design.

Interface requirements relate to support circuitry required to use the microprocessor with other devices in the system. A microprocessor that is very easy to use among the members of its own family of devices may turn out to be a horror to interface to the rest of the world. This is

particularly true of P-channel devices to be used in N-channel or TTL systems. Incompatibilities among system components can lead to problems and increased costs all over the system, including the previously mentioned clock and power supply areas.

6.1.6 Microprocessor Hardware Selection Summary

It should be obvious from the preceding brief discussion that picking microprocessor hardware is a tricky business. Even ignoring the software criteria, you must be very certain you get a device which will meet your system requirements at the lowest cost. It is important to remember at this point that lowest system cost may not always be the same as lowest possible hardware cost. Modification ease, maintenance and other factors may enter into the picture. There are times when you may want to knowingly allow some extra hardware cost to lower the costs in some other area. We will point out these areas as we go along.

6.2 Software Costs

Software costs are insidious. You can't see it, or feel it, or hear it, but software can break your microcomputer project faster than almost anything. As hardware systems and peripheral devices become more and more standard, more and more of the design-to-price burden is going to fall on the designer who has to design the software to hold these hardware blocks together.

Software is characterized by a very high development cost and a very low duplication cost. By way of example, IBM's software development of OS 360 (a very large and complex software project, to put it mildly) is estimated to have taken over 5000 man years of development time. However, the entire system can easily be duplicated and stored on $1000 worth of magnetic tape. As we said, duplication is cheap, development is expensive. This characteristic brings with it the following generalization: software for use in high volume products must be fixed. It is absolutely not possible to produce low cost custom software. Once you commit a program to ROM, don't consider changing the program unless you are prepared to change every other identical ROM in every other system. (Not to mention updating all

reference documentation.) The cost of custom software (unless you are in that business) is so high as to completely preclude it from volume systems. The software development cost very quickly completely overshadows the hardware cost.

Software exerts cost pressure on projects in two basic ways. The first is when poor technique and analysis lead to systems with inefficient use of expensive hardware resources. This causes the system to end up with more memory than it really needed, high speed interfaces that could have been eliminated with good software, extra I/O ports that some software multiplexing could have eliminated, and so on. The second way software raises cost is in the development/support cycle. This results in late projects due to inadequate time requirement forecasting, program bugs that turn up just after you take delivery on 10,000 mask programmed ROMs, documentation that requires a complete software system redesign when the program has to be changed a year after release, and other gory, expensive examples. Of the two areas software causes problems, the second is far more serious than the first. The first set of problems will naturally become less severe as you become more familiar with hardware/software system designs. (After all, that's what this course is here to teach you.) The second set belong to that group of problems that the entire computer community suffers along with year after year. Some progress is being made, but it is still a thorny problem. Good engineering practice is your best defense. Remember this basic rule: hardware and software design are equally complicated. The only difference is the rules.

Let's look at those areas where software can raise (or lower) your hardware costs. Remember we are considering a sliding cost scale from all hardware to virtually all software.

6.2.1 Processor Organization

The architecture of the processor you choose for your system can have a significant effect on your software costs. This is felt primarily in two areas: memory and I/O. A processor which is deficient in memory addressing modes will require larger programs to accomplish the same job as a

processor with more flexible addressing. More program means more memories, and more memories means more cost. A lack of on-chip registers may require you to use memory for temporary data storage. These memory references take more time during program execution and may make the difference between a simple (i.e. cheap) interface and a more complex (i.e. expensive) one. A versatile interrupt system may enable you to do most of the interrupt decoding with logic built into the CPU. Otherwise, you will have to add more service routines, I/O devices and money. A processor with a versatile instruction set may enable you to implement your programs much more efficiently, thereby saving memory space. The list goes on and on. Any area of the microprocessor's architecture can become a cost sensitive point in certain applications. The ultimate goal is to find the cost sensitive areas in your application and pick a processor that is strong in those areas.

6.2.2 Program Structure

The program structure, just as with the processor architecture, exerts its primary effect on the system memory requirements and I/O structure. Poorly designed programs will often take twice the memory of more carefully designed programs. You must balance the time and cost required to optimize programs against the cost of memory saved. Ideally, you will become skilled enough to design near optimum code the first time, thereby avoiding the expensive refinement procedure. Also, different program structures can be used to get maximum speed of program execution in speed sensitive areas. Failure to take advantage of these structures can result in the use of more expensive I/O interface hardware than is actually needed. The different program structures and their tradeoffs in speed and memory usage are discussed throughout the software lessons.

6.2.3 Implementation Language

The level at which you develop your programs has its primary effect on system memory size and overall system speed. Programs developed in higher level languages will generally be faster to develop, but they will take more time to execute and occupy from two to ten times more memory than the same program done in assembly language. Assembly language programs can

be designed for optimum memory usage and system speed but they take more time to develop. A data processing industry estimate is that assembly language programs take from two to five times longer to develop than comparable higher level language programs. This is particularly true of large, complex systems. You must balance the cost of development against the cost of the additional hardware resources. As a general rule, higher level languages will be lower in cost for small quantities of systems with assembly language becoming more cost effective as production quantity increases. (This assumes the higher level language programs can meet all system speed requirements without extra work.) The higher level language/assembly language tradeoffs are discussed in Lessons 9 and 10.

6.3 Systems Cost

Beyond the costs associated with producing the hardware are those costs associated with developing and maintaining the product. Unlike production costs, which are incurred as a function of how many units are produced, these costs are largely independent of production. Indeed, it is possible to incur very large costs in this area and never produce a single unit.

6.3.1 Development Costs

System Development Costs include all of the expenses you incur during the design of the product. Since these costs will be incurred prior to production, they will usually have to be met from your available resources. The areas of cost in this phase are all well known. However, the addition of software development adds a few extra categories.

Hardware Selection

All time and money spent evaluating various microprocessors and system components prior to commencing the actual system design. This would also include all analysis of crucial timing and interfaces and the initial partitioning of the system into hardware and software blocks.

Hardware Design

All time and money spent designing and debugging the hardware required to implement the system hardware.

Software Design

All time and money spent designing and debugging the programs required for use in the system. This may include a significant amount of expense for timesharing computer usage if you do not have the required program translation facilities available in house.

Development Tools

This includes any special hardware (such as a microcomputer development system or special test hardware) you have to buy for debugging and checking out the system design. Some of this cost will actually be spread out over all developments which end up based on the same microprocessor.

Documentation

All cost spent in developing the user manuals, production documents, reference specifications, and other documents essential to converting a working lab project into a viable product. This cost should not be underestimated. Thorough documentation will probably consume 20-25 percent of your development budget. However, it will be money well spent as your product matures and requires changes.

Marketing

This is the cost incurred in taking your finished product from the lab and presenting it to the world. This is not usually an engineering activity.

6.3.2 Modification Costs

Once you have a working product, there is always the possibility that you will want to issue a new, improved version. This is one area where a microprocessor based system can really save you time and money. In a total hardware system, a design extension or re-design will usually mean an almost total re-investment of the initial development costs. However, with a microprocessor based system you may be able to make substantial

functional changes with little or no changes to the hardware. This is because a software system can be re-configured by changing the program. Bearing in mind that all the software cost rules still apply, this is still usually a very effective technique. Expanding or changing an existing system is one area where you will find that the money spent on documentation was well spent. It can often make the difference between a successful and cost effective design modification or a complete re-design.

Program changes will often not be effective in products which were optimized so completely initially that there is not much extra hardware left to work with. The program can, after all, only perform functions which use available hardware. No matter how clever your programmer, if there isn't enough memory or I/O ports, some things just won't be feasible. If you have a product which looks like it is a candidate for later expansion, you may wish to incur a little higher production cost initially by adding some hardware for later use.

6.3.3 Maintenance Costs

Any cost you incur when your product fails in the field comes under this heading. All those field servicemen, return clerks, rework lines, and other support are expensive. Here too, the microprocessor can save you money. Almost by definition, the microprocessor must communicate with the entire system. This means that with the addition of some programming, memory, and some small amounts of hardware you can convert your microprocessor based system into its own diagnostic tester. You may not need to provide thorough tests, but even some simple tests can make troubleshooting a lot easier. Anything you can do to make testing and servicing easier will lower your maintenance costs.

Naturally, you must weigh the benefits of self-testing against the cost it will add. Often, however, you will discover at the end of the project that you have some extra I/O lines or a partially full ROM. Since these are going to be there anyhow, you may as well use them if you can. Since this type of thing is not usually discovered until well into the project, the addition of self test features at that point is one of the few times

when it may be desirable to add features after the design has started. However, if you want to be sure you have self-testing you should never wait to see what is left over. In that case, the self-testing features should be designed in like any other system feature.

6.4 A Perspective On Costs

Now that we have examined the various component costs, let's see how they relate to the total cost per unit of our proposed product. Over the total life of a product, the cost can be represented by the following general equation:

$$TC = \frac{FC}{N} + VC$$

where TC is the total cost per unit,
FC is the fixed cost required to develop and maintain the product,
VC is the variable cost associated with producing each unit, and
N is the number of units.

The terms in this equation can now be further broken down into those cost areas we discussed in the previous sections. Thus the fixed cost portions of the equation would turn out to be the development costs of the hardware and software, the documentation, the modification costs to the line of products, marketing, and all other cost which is incurred regardless of the volume of product produced. These costs are amortized over the number of units produced; the larger the number of units produced, the lower the fixed cost per unit.

The variable costs would be the cost of all the hardware components, production labor, field service for the percentage of units which prove defective, and all those other costs which vary based upon the number of units produced.

It is clear from this equation that the area where we will want to direct our cost reduction effort is dependent upon the quantity of units produced. For small quantities of units, we will want to minimize the fixed costs.

In practical terms this means using higher level languages (when available), hardware that is designed for ease of debugging and high reliability, and a general emphasis on development speed rather than low cost production. Conversely, for high volume production we will want to absolutely minimize production costs. This means highly optimized programs to minimize memory use, maximum use of program controlled interfaces to eliminate unneeded hardware, mechanical designs for easy production and any other techniques which can be used to hold the cost down.

The exact point at which the emphasis shifts from fixed cost reduction to variable cost reduction naturally changes for every product. In general, the more expensive the final product, the lower the emphasis on the variable costs.

6.5 Trading Off Software and Hardware

Now that we have discussed the main factors affecting system performance and cost, we can discuss the areas where system problems will force us to trade off hardware and software to modify system performance and cost. As we mentioned earlier, high speed (programmed, hardware, or whatever), large numbers of parts, and complex software are all expensive. We will be trying to implement all required system functions using the minimum cost combination of these items.

6.5.1 Conditions Which Lead to Design Trade Offs

In the course of the design we will be faced with several possible project conditions, some of which will require us to consider the various possible system tradeoffs. These conditions can be summarized as follows:

1. system speed too low, system cost too high,
2. system speed too low, system cost acceptable,
3. system speed acceptable, system cost too high,
4. system speed acceptable, system cost acceptable,
5. system speed excessive, system cost too high,
6. system speed excessive, system cost acceptable.

Clearly, each of these conditions requires different remedial action. Condition one is an obvious crisis situation. Unless some major breakthrough can be discovered, the project is probably doomed. Condition two is also fairly critical. It can be worked on only if the necessary speed can be acquired without driving cost into the unacceptable range. Very careful analysis will be required. Conditions three and five are probably both solvable by application of some hardware/software trade offs. Conditions four and six can be left alone. They may also be examined to see if extra features might be added to utilize the excess system speed without increasing the cost to an unacceptable level. If you elect to try this, be very careful not to go overboard. Any additions are best made in very small controlled increments. Avoid "creeping features" (see Lesson 2). If you aren't sure what to add, don't. Be happy you brought this one in under budget and save your money for next time.

After you figure out which condition your project is in, you have three alternatives: built it, change it, or cancel it. Building it or canceling it are decisions that you have to make on a situation by situation basis. Changing it may help you postpone that decision for awhile, but ultimately you will still have to decide. We can now examine how to change it so that hopefully you can decide to build it.

6.5.2 System Speed Problems

As we have emphasized all along, speed usually costs money. There are very few situations where increasing system speed lowers the cost. If you have a project which has to have increased speed, you might consider the title of this section to be "Trade Offs that Increase Cost". With that in mind, we can examine where to look to increase system speed.

System speed problems can be broadly divided into data transfer rate problems and data manipulation rate problems. In system operation these two types of problems will require distinctly different solutions. However, the same general techniques will apply to correcting both.

6.5.2.1 Data Transfer Rate Problems

Data transfer problems are encountered when transferring data between the computer and system I/O devices. This class of speed problem can be further subdivided into <u>processor rate limited problems</u> and <u>peripheral rate limited problems</u>. Processor rate limited problems arise when the computer is transferring data to a device which must have a high, non-varying transfer rate. This is characteristic of many real time interfaces, disk drives, and high speed buffered I/O devices. In the case of the disk drive, for example, it is not practical for the computer to vary the speed of disk rotation. Therefore, the processor must be able to read the data as fast as the rotating disk presents it to the read head. Data transfer rate problems of this type will result in lost or erroneous data. They represent the most serious system speed problems and they must be detected and corrected before the system will function properly.

Curing processor rate limited problems where the speed differential is excessive requires the addition of hardware to transfer some of the speed burden from the CPU. If the speed differential is close, restructuring the program sections which perform the actual data transfers may provide the speed margins you need. However, since instructions execute in fixed multiples of system cycle times, it will be impossible to adjust the system speed any more accurately than the execution time of the fastest instructions. For this type of problem, adjusting system speed by varying the program structure will only be effective over a fairly narrow range of timing.

Unlike processor rate problems, peripheral rate limited problems turn up when the computer is able to process the data at a much higher rate than the I/O devices can supply or accept it. This problem is most commonly encountered when the microcomputer is communicating with peripherals which are mechanical or which require user interaction, i.e. printers, tape readers, teletypewriters, etc. For example, many small microcomputer systems rely on the Teletype Corporation's model ASR 33 teletype as the main system peripheral. It serves as the keyboard, display, punch and reader for all program I/O operations. Now the teletype can only transfer

data at the rate of ten characters per second, or one data byte every 100 milliseconds. Printing 2500 characters (a small program listing) will take over four minutes. In this case, the computer will be spending most of its time waiting for the teletype to finish printing.

Peripheral rate problems are probably the most commonly encountered system speed problems. Fortunately, they seldom present a critical design problem. The cure is usually to add a faster I/O device. Even this solution has limitations. Most computer peripherals involve mechanical devices, and these will almost always be slower than the computer. You must trade off the cost of the faster peripheral against the time saved. If you discover you have a system which spends most of its time waiting for I/O transfers (a condition referred to as I/O bound), you may want to see if you can come up with some features to utilize what is essentially free processor time. Even better, you may be able to use some of that time to replace some hardware and further lower system cost. On the other hand, if the system can do everything it needs to at a cost you can afford, who cares if it spends 95% of its time waiting for the user to press a key? Microprocessor hardware is going to become so inexpensive that it will probably become far more economical to underutilize several microprocessors than to spend the development cost to optimize the use of one.

6.5.2.2 Data Manipulation Rate Problems

Where data transfer rate problems were related to how fast we can get data in and out of the computer, the data manipulation rate problems are concerned with how fast the data is processed once the computer has it. Where data transfer rate problems will be solved mainly be adding or changing system hardware, data manipulation rate problems will be solved mainly by restructuring the system's software.

The typical data manipulation problem arises when some section (or sections) of the system program takes an excessive amount of time to execute. The more commonly used that portion of the program, the worse the problem. This type of problem is characterized by your pushing a button and waiting for fifteen seconds until the teletypewriter prints the ten digit answer

to your equation. Using some hand held scientific calculators for complex calculations (try SIN 89°) provides some excellent examples of data manipulation rate limitations.

Some problems of this type are unavoidable in microprocessor systems. Their low speed (relative to minicomputers and large computers), modest instruction sets, and small data element size limit the efficiency with which any program will run. They are simply not designed for complex data processing applications. No matter how good the algorithm, certain classes of operations are going to take up significant amounts of computing time. Some examples of this group are complex mathematics routines (anything more complicated than a sixteen-bit integer divide can safely be considered complex), large memory searches, array operations, and moving blocks of data around in memory. In the large and minicomputer world, another primary cause of this problem is multiple user systems. Fortunately, to date the microprocessor world has been spared this particular problem. If your system requires any of these types of operations, you will end up paying some speed penalty. You will be able to minimize it to some extent, but it will be there. Fortunately, the types of applications which will use microprocessors do not normally require large numbers of complex operations. If you have one that does, you might seriously consider one of the sixteen-bit microprocessors or a low end minicomputer.

6.5.3 System Cost Problems

System cost problems become significant when you have a working system which must be made more economical for practical production. The term "problems" in this context is probably misleading. Virtually all systems intended for high volume production will go through some cost optimization procedure between prototype and final production. Usually you will have decided that the cost range for the product is acceptable before proceeding with the development. This decision is based on market studies, comparison with existing products, and other evaluations of what is a reasonable final selling price of the product. This number can then be projected back to arrive at a cost range for the product.

In general, the techniques for lowering product cost will be the reverse of techniques to increase speed. You will want to remove extraneous hardware, compact all programs into minimum memory space, and in general, make the maximum use of the processor and software to implement system functions. This must all be done without creating any system speed problems. Therefore, the proceedure is best carried out in discrete steps. You refine one section of the system, make sure the system still works, and move on to the next section. Ultimately you will reach a point where no further cost economies can be achieved without compromising system performance.

Cost optimization should always be undertaken with the firm realization that the end must justify the means. It is an expensive process that is usually only vigorously applied to products whose high volume will justify the expense. Otherwise the cost of the optimization will overshadow any savings made in production.

6.6 Hardware Speed Trade Offs

When you must modify system speed using hardware, you will be trying to either increase or decrease the amount of work done by the processor. In the first case you will be trying to simplify the system hardware or replace much of it with software. This results in decreased hardware cost and lower system speed. In the second case you will be trying to transfer some of the work being performed by the software out to the hardware. This will result in higher system cost. Within this framework let's examine some of the alternatives available.

6.6.1 Processors and Memories

A simple solution to some system speed problems may be to change processors within the same family. Some manufacturers provide microprocessors which are graded by speed. If the nominal processor speed is 2 MHz, some devices may be available in selected speed ranges from 1 to 4 MHz. Since the processor cost goes up with the speed, using this method you only have to pay for the speed you require.

If you are considering a faster (or slower) processor, you must also consider the effect that memory speed has on program execution. The computer must get all instructions and data from memory. If the memory is not at least as fast as the processor, there is no point in increasing processor speed. Similarly, you may be able to increase system speed by using the same processor with faster memories.

6.6.2 Decode Logic

Decode logic is required for a variety of purposes in a microcomputer system. Most decoding is done to determine I/O device addresses and memory addresses. This logic is almost all done with hardware, and it can usually be minimized in a dedicated system. For example, many microprocessors can address 65K bytes of memory using 16 address lines. Very few applications will require this much memory, so after you determine how much memory the system requires, you can eliminate the excess decoding. For example, if you only need 4096 bytes of memory, you need only decode 12 address lines to access all valid memory addresses in your system. Similar minimization can be applied to the I/O device addresses.

One added benefit of reducing the decoding is that the undecoded lines can be used as extra control lines in the system. Usually the full address bus runs everywhere in the system. If system speed permits, the undecoded address lines may be used to eliminate further hardware control logic. In the case of the system with 4096 bytes of memory we mentioned earlier, the four unused address lines could be used individually (or even decoded) to provide system control signals. Similar trade offs can be performed in systems which require fewer I/O devices than the maximum available.

6.6.3 Memory Buffers

Memory buffers are used to collect or hold data that is in transit between the CPU and system peripheral devices. The addition of a high speed buffer dedicated to a specific peripheral can be used to solve processor data transfer rate problems. This is particularly effective if the peripheral has a low average data rate with high speed burst transfers of data. A

buffer can be used to collect the data during the burst transmission, with the CPU reading the individual data elements from the buffer after the transmission is complete. This type of buffering can also be used in conjunction with the computer's DMA facility. In this case, the buffer accumulates the data and transfers it into the main computer memory in a single block transfer.

Buffers can also be used to solve peripheral data rate problems. In this case, the CPU transfers the data out to the peripheral buffer. The peripheral can then take the characters at its own rate with no further processor intervention.

Addition of buffers to the system requires the addition of considerable hardware expense. Accordingly, they should only be added if the system really needs them. As long as speed is not a problem, most microprocessors can do a good job of implementing buffers. They can do this using already present main memory and some programming. Data is transferred into and out of this type of buffer using an interrupt. The device interrupts when it is ready for a transfer and the CPU performs a single transfer. When the buffer becomes full or empty, the data is then processed, just as with a dedicated buffer. This is always much cheaper than an external buffer system. In the course of the design, if you think you need data buffering, look very carefully to see if it can be done using software. Even after the design is done you may discover that a hardware buffer initially thought necessary can actually be done in this way. It may be worth the redesign cost to save the hardware cost, particularly if production volume will be high.

6.6.4 Specialized Interface Devices

A specialized interface device is designed to perform some defined function in the system. Usually the function to be performed could be performed using either software or the specialized device. You will consider a trade off when you either find yourself with a speed problem and no interface device or the interface device and lots of program time available. In the first case you design in the device to free up the program time that

performing the function is tying up. In the second case you take out the device and replace the function with software.

A common example of this type of device is the UART (Universal Asynchronous Receiver Transmitter). This device accepts parallel data and converts it to a serial bit stream conforming to the EIA RS232C data transmission standard. The function can easily be performed under program control, but as mentioned earlier, each character sent or received will take up 100 milliseconds of computer time. During this time the software must convert a character from parallel to serial, add start and stop bits and generate all timing and control signals required to perform the transfer. If your system has the time, fine. If it doesn't, you add a UART. The only time required now is the time required to write one parallel byte out to the UART. After that, the UART generates all those functions that were done by the software, freeing your processor to do other things. Similar trade offs can be made using other pre-defined functional devices.

6.6.5 Interrupts

In many systems the computer must spend considerable time responding to interrupts. If there is more than one possible interrupting device, the processor must determine which device generated the interrupt before it can process any data. This identification can be done in a combination of hardware and software that can be varied to meet system speed/cost requirements.

For maximum system speed you design the hardware so that each interrupting device responds to CPU acknowledgement with the address of its own dedicated service routine. This gives maximum response speed, since no time is spent decoding any device identification codes. In some processors this can be reduced to the interrupting device providing an actual subroutine call instruction, making the interrupt almost transparent in terms of overhead time loss.

To lower hardware expense, the device identification can be moved into the service routines. In this case, the interrupting devices all provide

the same routine address. The software must then poll all devices in the system to see who generated the interrupt. This adds a significant amount of overhead time to the routine, and will probably not be satisfactory for faster devices.

As a compromise, the system can be implemented as a combination of direct and indirect interrupt decoding. In this case, you assign your highest priority or fastest (usually the same) devices their own identification address. They will then interrupt directly to their routines with minimum time loss. The lower priority devices can then be assigned to a common address and these can be decoded under slower, cheaper software control.

6.7 Software Trade Offs

Software trade offs are made for the same reason as hardware trade offs, namely modifications of system cost and speed. Where we traded off hardware for different hardware or a combination of less hardware and some software, with software we will usually be trading off program speed for memory size. Increases in program speed will often take more memory, thereby costing more money. Conversely, if speed is not a problem, certain program types can be replaced by markedly less code, with a subsequent lowering of memory size and cost. It must be kept in mind, however, that not all decreases in program size lower memory cost nor do all increases in program size increase cost. The only time changes in program size affect memory cost at all is when the change results in the saving or use of an entire memory. For example, if your program is to be located in 2K x 8 mask programmed ROMs, the only time that your cost will change is when your program size exceeds multiples of 2048 bytes. Up to that point, the memory is essentially free. Similarly, if you discover your new, improved, program is now 2075 bytes long, you may want to expend some time eliminating those 27 extra bytes. (The terms and techniques discussed in the next few sections are covered in greater detail in the software lessons.)

6.7.1 Program Loops and Subroutines

Program loops and subroutines are used to minimize program size and control execution. A sequence of operations which is to be executed a fixed number

of times can be placed in a loop. A section of code common to several portions of the program can be placed in a subroutine. The actual coding is thereby only written one time no matter how many times the loop is executed or the subroutine is called. Loops and subroutines minimize program size at the expense of some program speed.

The instructions which must be executed to control execution of the loop or the calling of the subroutine take a certain amount of time that is not required for the actual function being performed. In speed critical situations the effect of these overhead instructions can be eliminated or modified to increase execution speed. This is done by replacing the loop or subroutine with the actual straight line code that was originally there. This eliminates the overhead instructions completely. Alternatively, a loop may be modified to use a lower percentage of its time for overhead. This is done by partially replacing the loop with the straight line code and lowering the number of times through the loop. For example, say a certain function is to be performed 10 times, once for each execution of the loop. In this case, say loop overhead is 20%. By duplicating the function and lowering the loop count to five we would do the same processing with only 10% overhead. The price would be a doubling of the amount of memory occupied by the function.

6.7.2 Functional Computations

Throughout your program you will use functional computations to evaluate data and decide on program responses to input conditions. You will be able to vary the execution speed and memory usage of many of these blocks based on how you evaluate the data. For example, let's say we have an application where we need to multiply two eight-bit integers. One solution is to write an algorithm which will multiply the two numbers. If for some reason the speed of the algorithm execution was not adequate for our application, we might consider storing all possible results in a ROM (or part of a ROM). We would then use our two numbers to compute the address of the product, thus removing most of the computations. This method should execute considerably faster. Again, the price is more memory usage.

In practice, not many mathematical functions can be produced in the manner just described. However, the technique is very often applicable to memory addresses. A required address can often be computed as part of the program execution or stored as fixed data. Computation by algorithm is more efficient, but fetching defined data is faster. These types of alternatives can be traded off throughout the course of system software design.

6.7.3 Repeated Computations

Related to functional computations is the class of program operations called repeated computations. Analysis of programs over the years has shown that in most programs 90% of the execution time is spent executing 10% of the program. These software "critical paths" are what we call repeated computations. If your system has a speed problem, the first thing to do is to see if you have any repeated computations. You can then devote your optimization effort in those areas where it will do the most good. Some common types of repeated computations are common mathematical functions, table searches, data movement routines, and data sorts.

If you find you have a clearly defined repeated computation, you may find it worthwhile to study it. See if you can find a better algorithm in the data processing literature. If you can't find one, do your best to devise one. Time spent thoroughly optimizing a repeated calculation can be far more valuable than partially optimizing several sections of less frequently executed code.

6.8 Summary

The hardware/software design procedure is something that you only learn by practice. You must gain first hand experience in the real world. It is a process which becomes more than designing the hardware and then designing the software. It is an integrated proceedure which will allow you to implement some of the most creative digital systems ever imagined. We have only scratched the surface of what is available, and what is available is just the beginning.

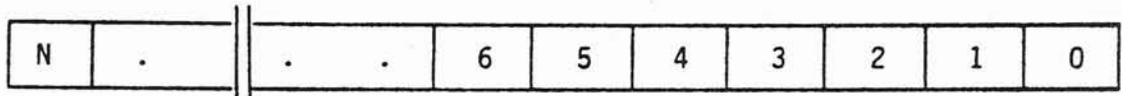
REPRESENTING BINARY DATA

REPRESENTING BINARY DATA

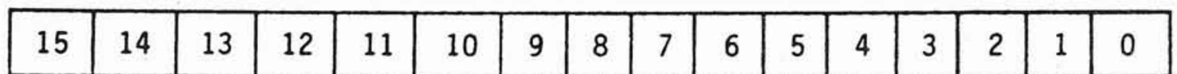
When working with digital computers it is necessary to work with binary data. Computer components are built up from electronic devices which can only represent data as 0's and 1's. This means you will have to use binary to represent numbers. In spite of this, it is impossible to escape from the fact that binary data is not overly convenient to use. We have all used base 10 numbers for years and the base 2 number system seems quite inefficient by comparison. It takes 6 binary digits to represent the number 50_{10} (110010_2), and it gets worse. In this section we will discuss how the individual binary data elements are represented and how they can be grouped together for more convenient use. Binary arithmetic and logic are discussed in the following supplementary section.

7.1 - Binary Data Elements

A computer data element of arbitrary length N is shown below.

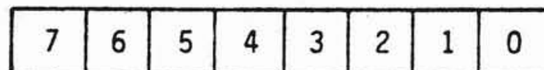


The right most bit (bit 0) is considered to be the least significant bit. Bit N, at the left most position, is considered to be the most significant bit. Thus a computer with a 16-bit data element would have data in bit positions 0-15:



16-Bit Data Word

Similarly, an 8-bit microcomputer would have data in bit positions 0-7:



8-Bit Data Word

Thus when we speak of loading one's into bits five and seven of an eight-bit register, we will be loading the following pattern.

7	6	5	4	3	2	1	0	Bit Position
1	0	1	0	0	0	0	0	

References to the bit positions of a register or memory location rather than to the binary number in a register or memory location are common in control and logic applications.

7.2 - Binary Numbers

All number systems (including binary) represent numbers as a function of the radix (number base) and the position of the individual digits. Any number composed of digits A_N-A_0 in a radix R can be represented as follows:

$$A_N A_{N-1} \ . \ . \ . \ A_1 A_0 = A_N R^N + A_{N-1} x R^{N-1} + \ . \ . \ . \ . \ + A_1 x R^1 + A_0 x R^0$$

where A is any digit in the range 0 to R-1 and N is the digit position. For example, consider the number 136 in base 10. We have digits in positions 0, 1 and 2. In this case, all values of A must be in the range 0-9, and R = 10. We thus have a number represented as

$$136_{10} = 1 \times R^2 + 3 \times R^1 + 6 \times R^0$$

(subscript to identify number base.)

$$= 1 \times (10)^2 + 3 \times (10) + 6 \times 1$$

$$= 100 + 30 + 6$$

$$= 136_{10}$$

Binary numbers can be similarly represented. The difference is that where in decimal we have ten possible numbers (0-9), in binary we only have two (0 and 1). This means that representing a given number in binary will require more digit positions than representing the same number in decimal. Thus the binary number 10110 is represented as

$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

$$= 1 \times 16 + 0 + 4 + 2 + 0$$

$$= 22_{10}$$

NUMBER SYSTEM CONVERSIONS

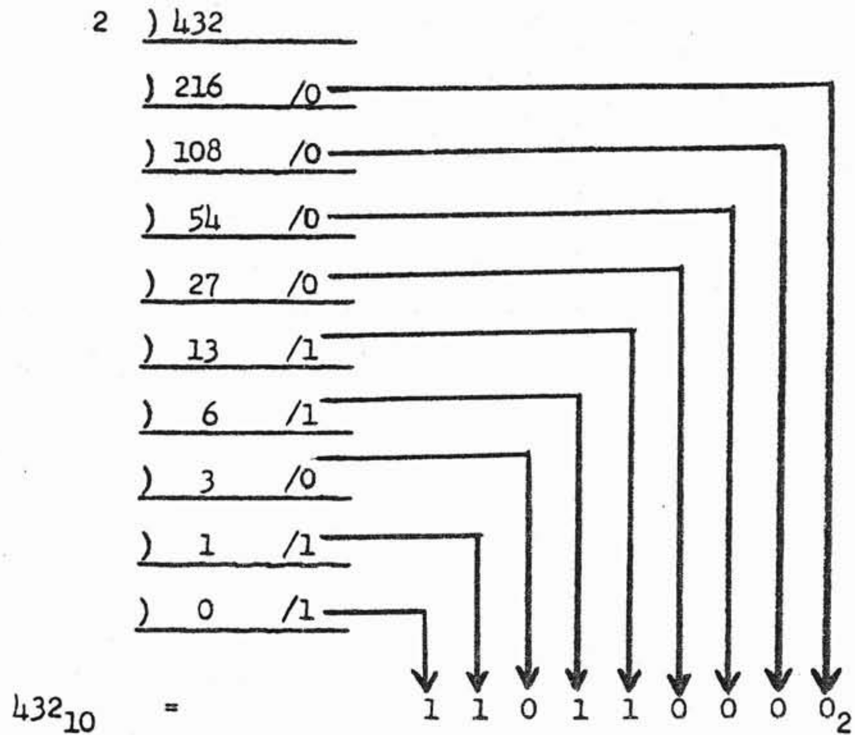
NUMBER SYSTEM CONVERSIONS

DECIMAL TO BINARY

To convert any decimal number to a binary number, take the decimal number and successively divide by "2" and write down the remainder (1 or 0) as you continue dividing until the number becomes "0".

EXAMPLE:

Convert 432_{10} to a binary number.



432_{10} = $1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0_2$

BINARY TO DECIMAL

As in the decimal number system, the least significant digit is on the right and the most significant digit is on the left and each digit is a multiple of a certain power of 10.

$$432_{10} = 4 \times 10^2 + 3 \times 10^1 + 2 \times 10^0$$

This is also true for a binary number, except that it is a multiple of a certain power of "2".

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

So to convert a binary number to a decimal number, take each power of "2" and change it to its respective decimal number.

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$2^5 = 32$

.

.

.

$2^{16} = 65,536$

etc.

EXAMPLE:

Take the number 101101

$101101_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$

$= 32 + 0 + 8 + 4 + 0 + 1$

$101101_2 = 45_{10}$

CONVERTING DECIMAL TO OCTAL

Use the same method as to convert decimal to binary except divide by the base of "8" instead of "2".

EXAMPLE:

Convert 131_{10} to Octal

```
8 ) 131
  )  16   /3
  )   2   /0
  )   0   /2
```

$131_{10} = 203_8$

CONVERTING OCTAL TO DECIMAL

To convert from octal to decimal use the same method as for binary to decimal conversion, except use the powers of "8" instead of "2".

EXAMPLE:

$8^0 = 1$

$8^1 = 8$

$8^2 = 64$

$8^3 = 512$

$8^4 = 4096$

$8^5 = 32768$

$8^6 = 262144$

etc.

EXAMPLE:

Convert 203_8 to decimal

$$
\begin{aligned}
203_8 &= 2 \times 8^2 + 0 \times 8^1 + 3 \times 8^0 \\
&= 2 \times 64 + 0 \times 8 + 3 \times 1 \\
&= 128 + 0 + 3 \\
&= 131
\end{aligned}
$$

CONVERTING DECIMAL TO HEXADECIMAL

Again, use the same method for decimal to binary conversion, except use the base "16" instead and replace the remainders of "10" to "15" by the letters A to F respectively.

10 = A

11 = B

12 = C

13 = D

14 = E

15 = F

EXAMPLE:

Convert 9192_{10} to hexadecimal

16	) 9192		
	) 574 / 8	=	8
	) 35 / 14	=	E
	) 2 / 3	=	3
	) 0 / 2	=	2
	9192_{10}	=	$23E8_{16}$

CONVERTING HEXADECIMAL TO DECIMAL

Again, use the same method for binary to decimal conversion, except use the powers of 16 instead, and convert the letters A through F to 10 through 15 respectively.

$$16^0 = 1$$
$$16^1 = 16$$
$$16^2 = 256$$
$$16^3 = 4096$$
$$16^4 = 65536$$

EXAMPLE:

Convert $23E8_{16}$ to decimal

$$23E8_{16} = 2 \times 16^3 + 3 \times 16^2 + E \times 16^1 + 8 \times 16^0$$
$$= 2 \times 16^3 + 3 \times 16^2 + 14 \times 16^1 + 8 \times 16^0$$
$$= 2 \times 4096 + 3 \times 256 + 14 \times 16 + 8 \times 1$$
$$= 8192 + 768 + 224 + 8$$
$$23E8_{16} = 9192_{10}$$

CONVERTING OCTAL TO BINARY, HEXADECIMAL TO BINARY, OCTAL TO HEXADECIMAL; AND BACK

Convert to binary first, then **if needed,** regroup the binary numbers into the desired groups of three or four binary digits, (three for octal or four for hexadecimal). These translate directly to the desired number system. Always start the regrouping with the LSB.

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

EXAMPLE:

1) Convert $4A2BC_{16}$ to Octal

$4A2BC_{16}$ = 0100 1010 0010 1011 1100

= 01/00 1/010 /001/0 10/11 1/100

= 01 001 010 001 010 111 100

= 1 1 2 1 2 7 4

$4A2BC_{16}$ = 1121274_8

2) Convert 1435_8 to Hexadecimal

1435_8 = 001 100 011 101

= 001 1/00 01/1 101

= 0011 0001 1101

= 3 1 D

$1435_8 = 31D_{16}$

BCD NUMBERS

BCD NUMBERS

In some applications it is desirable to be able to directly represent decimal numbers in the binary computer. This is done using Binary Coded Decimal, BCD. When using BCD we do not use all possible data values that the binary data element can represent. Instead, we limit ourselves to the following four bit patterns:

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The other six four bit combinations (1010-1111) are not used in BCD.

An eight-bit data element can hold two BCD digits. This means it can represent decimal numbers from 0-99. BCD is very commonly encountered in control and instrument interface applications. As a result, many computers provide instructions to allow direct arithmetic with BCD numbers.

BINARY FRACTIONS

BINARY FRACTIONS

Binary numbers are generally considered as whole integers (i.e., 1, 2, 3, ...). However, it often becomes necessary to represent numbers other than whole numbers. Binary fraction representation is analagous to decimal fraction representation. In decimal numbers a fraction consists of digits to the right of a decimal point; in binary, we consider the bits to the right of the binary point to be a fraction. In a binary fraction the bits represent 2^{-N}, where N = the bit position to the right of the binary point. The powers of 2^{-N} are shown in the number tables. Consider the following binary number:

Binary Point

1 1 0 1 . 1 1 0 1

This binary number representation means

$$2^3 + 2^2 + 0 + 2^0 + 2^{-1} + 2^{-2} + 0 + 2^{-3}$$

$$= 8 + 4 + 1 + .5 + .25 + .0625$$

$$= 13.8125$$

Numbers can be converted to and from binary fractions using the techniques already shown for converting whole binary numbers. Unfortunately, not all fractions are as well behaved as the above example. Consider the decimal number 3 1/3. When we try to convert it we end up with -

$$3\ 1/3_{10} = 3.3333......3_{10}$$

$$3\ 1/3_{10} = 11.01010101......01_2$$

The fraction repeats and there is obviously no exact result. We will have to choose a bit position where we truncate the value. For example, if we choose bit position six, we end up with an approximation.

$$3\ 1/3_{10}\ N\ 1\ 1\ .\ 0\ 1\ 0\ 1\ 0\ 1$$

The rounding error introduced by this truncation can be computed by converting the truncated fraction.

$$11.01010101 = 2 + 1 + .25 + .0625 + .015625$$
$$= 3.328125$$

The error is about .1%. The possibility of this type of rounding error must always be taken into consideration when using binary fractions, particularly in division operations. Very few numbers result in exact binary fractions, so the possibility of error will be ever present.

BINARY ARITHMETIC AND LOGIC INSTRUCTIONS

BINARY ARITHMETIC AND LOGIC INSTRUCTIONS

All computers provide a number of instructions which are used to perform arithmetic and logic operations on data. As discussed elsewhere, computer data consists of patterns of bits in registers and memory locations. However, there is a fundamental difference between the way the arithmetic instructions and logic instructions treat this binary data. The arithmetic instructions interpret the data as numbers. The logic instructions, on the other hand, interpret the data as a collection of individual bits.

11.1 Computer Arithmetic Instructions

The most basic computer arithmetic instruction is addition. This instruction and the logic complement instruction can be used to implement any known mathematical function. As a result, many computers offer addition as their only arithmetic instruction. After addition, the next most common arithmetic instruction is subtraction. This is because subtraction can be performed using the same basic hardware as addition. After addition and subtraction you have to go to a considerably more complex computer to get multiplication and division as built-in functions. The hardware required for these operations is considerably more complex than that used for addition and subtraction. As of this writing (July 1976) there are no microprocessors with built-in multiply and divide hardware. This will certainly change. All of these operations use the contents of the computer's accumulator(s) and another data source as operands with the result ending up in the accumulator.

11.1.1 Twos Complement Notation

The most common way of representing numbers for arithmetic operations in the computer is twos complement notation. To understand twos complement notation let's consider the binary numbers that can be represented by an 8-bit data element. We know that an 8-bit data element can represent 256 individual values. When we use a register as a counter, we can count up to 256 different values. The binary number counting method follows.

Binary	Count
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
.	.
.	.
.	.
11111110	254
11111111	255

This is an example of using the data element as an unsigned number. The numbers in the register are interpreted as being in the range +0 to +255. Now this method is convenient for counting, but awkward for arithmetic because there is no way to represent negative numbers. To circumvent this problem we change the way in which we interpret the 256 possible numbers in the registers so that we will be able to represent both positive and negative numbers. This revised scheme will be called signed twos complement.

Any binary number is converted to its negative by complementing it, adding one, and ignoring any carry out causd by the addition. For example, consider the binary number +5 in an 8-bit microcomputer; 00000101. To convert +5 to -5 we first complement 11111010, and then add 1; 11111011.

If we perform the procedure on the result we get back our original number. The 256 possible numbers in an 8-bit register now represent positive and negative numbers in the range -128 to +127 as follows:

```
0 0 0 0 0 0 0 0 = 0
0 0 0 0 0 0 0 1 = 1
        .       .
        .       .
        .       .
0 1 1 1 1 1 1 1 = +127
1 1 1 1 1 1 1 1 = -1
1 1 1 1 1 1 1 0 = -2
1 1 1 1 1 1 0 1 = -3
        .       .
        .       .
        .       .
1 0 0 0 0 0 0 0 = -128
```

Note how bit 7 (the most significant bit) is always a zero for all positive numbers and always a one for negative numbers. The most significant bit in a twos complement number is called the sign bit, because by testing it you can determine if a number is positive or negative.

11.1.2 Binary Arithmetic

Binary arithmetic is performed using the ALU and two operands. The operation can be performed using either unsigned numbers or signed twos complement numbers, depending upon the operation being performed. Most computers perform addition and subtraction as unsigned operations. They do provide flags to indicate the result in signed twos complement, but it is up to you to keep track of the sign and magnitude.

Addition is performed by adding the contents of an operand to the contents of the accumulator. If the result is greater than the largest number which can be represented in the accumulator, a flag will be set to indicate a carry out has occurred. For example, consider the operation of adding the number 15_{10} to an 8-bit accumulator which contains 25_{10}. The operation would be performed as follows.

```
        Accumulator   0 0 0 1 1 0 0 1
          + Operand   0 0 0 0 1 1 1 1
             Result   0 0 1 0 1 0 0 0 = 40₁₀
```

Now consider the addition of 111_{10} to 145_{10} in the accumulator.

```
        Accumulator   1 0 0 1 0 0 0 1
          + Operand   0 1 1 0 1 1 1 1
                   1/ 0 0 0 0 0 0 0 0
             carry out
```

The result of this operation is 256_{10}. It causes a carry out to indicate that the accumulator overflowed.

Subtraction is performed by taking the twos complement of the subtrahend and adding it to the minuend in the accumulator. Thus to subtract 10_{10} from 25_{10} we would perform the following operation.

```
             Subtrahend   0 0 0 0 1 0 1 0
   Form Twos Complement   1 1 1 1 0 1 1 0
     Add To Accumulator   0 0 0 1 1 0 0 1
                       1/ 0 0 0 0 1 1 1 1 = 15₁₀
                       carry out
```

Ignoring the carry out, we have a result of 15_{10}. Now consider the subtraction of 35 from 15.

```
             Subtrahend   0 0 1 0 0 0 1 1
   Form Twos Complement   1 1 0 1 1 1 0 1
     Add To Accumulator   0 0 0 0 1 1 1 1
                       0/ 1 1 1 0 1 1 0 0
                       no carry
```

No carry indicates a negative result. If we convert the number using our twos complement rules, we obtain the correct result, -20.

A minuend
− B subtrahend
R result

Result	1 1 1 0 1 1 0 0
Complement	0 0 0 1 0 0 1 1
Add 1	0 0 0 1 0 1 0 0 = 20_{10}

Notice that the sense of the carry after a subtraction is reversed from that of addition. A carry out indicates that the subtrahend was smaller than the minuend and the result of the subtraction was positive. No carry out indicates that the subtrahend was larger than the minuend and that the result was negative. This is called a borrow condition, and it is analagous to overflow in an addition operation. To avoid confusion about the reversal of the state of the carry flag, many computer ALU's automatically complement the carry flag after a subtraction. This makes its state after a subtraction match more closely its state after an addition (i.e. carry set if result cause a borrow, clear if the result did not cause a borrow).

11.2 Computer Logic Instructions

In contrast to the arithmetic instructions, the logic instructions perform their operations with no regard for the number representation being used. The numbers being operated upon are simply treated as strings of bits. That is why these operations are often referred to as bit by bit operations. The operation performed on one bit in no way affects the operation upon adjacent bits.

The four most common computer logic instructions are Complement, AND, OR, and Exclusive OR. These operations (except complement) use the contents of the accumulator and another data source as operands, with the result ending up in the accumulator.

11.2.1 Logic Complement

The complement instruction replaces each bit in the accumulator with its logic complement. Thus if the accumulator contains 1 0 1 0 1 1 0 1, the complement operation yields the following result.

Accumulator	1 0 1 0 1 1 0 1
Complement	0 1 0 1 0 0 1 0

11.2.2 Logic AND

The Logic AND operations (Symbol Λ) operates upon the bits of the accumulator and an operand according to the following truth table.

Accumulator Bit	0 0 1 1
Operand Bit	0 1 0 1
Result Bit	0 0 0 1

Thus only those bit positions which are logic ones in both the accumulator and the operand will be logic ones in the accumulator after a Logic AND operation has been performed. For example, consider the following Logic AND operation.

Accumulator	0 1 1 0 1 1 0 1
Λ Operand	1 1 0 1 1 0 1 1
Result	0 1 0 0 1 0 0 1

Only those bits which were ones in both operands are in the result.

11.2.3 Logic OR

The Logic OR operation (Symbol V) operates upon the bits of the accumulator and an operand according to the following truth table.

Accumulator Bit	0 0 1 1
Operand Bit	0 1 0 1
Result Bit	0 1 1 1

Bit positions which are logic ones in either the accumulator or the operand will be Logic ones in the accumulator after a Logic OR operation has been performed. For example, consider the following Logic OR operation.

Bit positions which are Logic ones in either the accumulator or the operand will be Logic ones in the accumulator after a Logic OR operation has been performed. For example, consider the following Logic OR operation.

Accumulator	1 0 1 1 0 1 1 0
V Operand	0 0 1 1 0 0 1 1
Result	1 1 1 1 0 1 1 1

All bits which were ones in both operands are ones in the results.

11.2.4 Logic XOR

The Logic Exclusive OR operation (Symbol A, o-ten called XOR) is not found in all computers. It operates upon the bits of the accumulator and an operand according to the following truth table.

Accumulator Bit	0 0 1 1
Operand Bit	0 1 0 1
Result Bit	0 1 1 0

Bit positions which are a Logic one in either the accumulator or the operand but not both will be Logic ones in the accumulator after a logic XOR operation has been performed. For example, consider the following Exclusve OR operation.

Accumulator	0 1 1 0 0 1 0 1
⊻ Operand	1 0 1 1 0 1 1 0
Result	1 1 0 1 0 0 1 1

Those bits which were ones in only one of the operands are ones in the result.

APPENDIX A

MODIFIED 6500 OP CODE TABLE

MODIFIED 6500 OP CODE TABLE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK	ORAxi	*	*	*	ORAz	ASLz	*	PHP	ORA#	ASLa	*	*	ORA@	ASL@	*	0
1	BPLr	ORAiy	*	*	*	ORAzx	ASLzx	*	CLC	ORA@y	*	*	*	ORA@x	ASL@x	*	1
2	JSR@	ANDxi	*	*	BITz	ANDz	ROLz	*	PLP	AND#	ROLa	*	BIT@	AND@	ROL@	*	2
3	BMIr	ANDiy	*	*	*	ANDzx	ROLzx	*	SEC	AND@y	*	*	*	AND@x	ROL@x	*	3
4	RTI	EORxi	*	*	*	EORz	LSRz	*	PHA	EOR#	LSRa	*	JMP@	EOR@	LSR@	*	4
5	BVCr	EORiy	*	*	*	EORzx	LSRzx	*	CLI	EOR@y	*	*	*	EOR@x	LSR@x	*	5
6	RTS	ADCxi	*	*	*	ADCz	RORz	*	PLA	ADC#	RORa	*	JMPi	ADC@	ROR@	*	6
7	BVSr	ADCiy	*	*	*	ADCzx	RORzx	*	SEI	ADC@y	*	*	*	ADC@x	ROR@x	*	7
8	*	STAxi	*	*	STYz	STAz	STXz	*	DEY	*	TXA	*	STY@	STA@	STX@	*	8
9	BCCr	STAiy	*	*	STYzx	STAzx	STXzy	*	TYA	STA@y	TXS	*	*	STA@x	*	*	9
A	LDY#	LDAxi	LDX#	*	LDYz	LDAz	LDXz	*	TAY	LDA#	TAX	*	LDY@	LDA@	LDX@	*	A
B	BCSr	LDAiy	*	*	LDYzx	LDAzx	LDXzy	*	CLV	LDA@y	TSX	*	LDY@x	LDA@x	LDX@y	*	B
C	CPY#	CMPxi	*	*	CPYz	CMPz	DECz	*	INY	CMP#	DEX	*	CPY@	CMP@	DEC@	*	C
D	BNEr	CMPiy	*	*	*	CMPzx	DECzx	*	CLD	CMP@y	*	*	*	CMP@x	DEC@x	*	D
E	CPX#	SBCxi	*	*	CPXz	SBCz	INCz	*	INX	SBC#	NOP	*	CPX@	SBC@	INC@	*	E
F	BEQr	SBCiy	*	*	*	SBCzx	INCzx	*	SED	SBC@y	*	*	*	SBC@x	INC@x	*	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Key to addressing symbols:

#	immediate	@x, @y	absolute indexed
@	absolute	zx, zy	zero page indexed
a	accumulator	xi	indexed indirect
r	relative	iy	indirect indexed
z	zero page		

APPENDIX B

KIM INFORMATION

KIM PROGRAMMING DATA SHEET

6500 OP CODE TABLE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK	ORAxi	*	*	*	ORAz	ASLz	*	PHP	ORA#	ASLa	*	*	ORA@	ASL@	*	0
1	BPLr	ORAiy	*	*	*	ORAzx	ASLzx	*	CLC	ORA@y	*	*	*	ORA@x	ASL@x	*	1
2	JSR@	ANDxi	*	*	BITz	ANDz	ROLz	*	PLP	AND#	ROLa	*	BIT@	AND@	ROL@	*	2
3	BMIr	ANDiy	*	*	*	ANDzx	ROLzx	*	SEC	AND@y	*	*	*	AND@x	ROL@x	*	3
4	RTI	EORxi	*	*	*	EORz	LSRz	*	PHA	EOR#	LSRa	*	JMP@	EOR@	LSR@	*	4
5	BVCr	EORiy	*	*	*	EORzx	LSRzx	*	CLI	EOR@y	*	*	*	EOR@x	LSR@x	*	5
6	RTS	ADCxi	*	*	*	ADCz	RORz	*	PLA	ADC#	RORa	*	JMPi	ADC@	ROR@	*	6
7	BVSr	ADCiy	*	*	*	ADCzx	RORzx	*	SEI	ADC@y	*	*	*	ADC@x	ROR@x	*	7
8	*	STAxi	*	*	STYz	STAz	STXz	*	DEY	*	TXA	*	STY@	STA@	STX@	*	8
9	BCCr	STAiy	*	*	STYzx	STAzx	STXzy	*	TYA	STA@y	TXS	*	*	STA@x	*	*	9
A	LDY#	LDAxi	LDX#	*	LDYz	LDAz	LDXz	*	TAY	LDA#	TAX	*	LDY@	LDA@	LDX@	*	A
B	BCSr	LDAiy	*	*	LDYzx	LDAzx	LDXzy	*	CLV	LDA@y	TSX	*	LDY@x	LDA@x	LDX@y	*	B
C	CPY#	CMPxi	*	*	CPYz	CMPz	DECz	*	INY	CMP#	DEX	*	CPY@	CMP@	DEC@	*	C
D	BNEr	CMPiy	*	*	*	CMPzx	DECzx	*	CLD	CMP@y	*	*	*	CMP@x	DEC@x	*	D
E	CPX#	SBCxi	*	*	CPXz	SBCz	INCz	*	INX	SBC#	NOP	*	CPX@	SBC@	INC@	*	E
F	BEQr	SBCiy	*	*	*	SBCzx	INCzx	*	SED	SBC@y	*	*	*	SBC@x	INC@x	*	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

STATUS REGISTER

```
P: N V   B D I Z C
   7 6 5 4 3 2 1 0
```

IMPORTANT ADDRESSES

00EF	PCL	17FA	NMI-L ss=00
00F0	PCH	17FB	NMI-H ss=1C
00F1	P	17FC	RST-L
00F2	S	17FD	RST-H
00F3	A	17FE	IRQ/BRK-L
00F4	Y	17FF	IRQ/BRK-H
00F5	X		
		00F1=	00 (LD)
1700	PAD	17F5	SAL
1701	PADD	17F6	SAH
1702	PBD	17F7	EAL+1
1703	PBDD	17F8	EAH
		17F9	ID#
		1800	DUMPT
		1873	LOADT

(handwritten notes: "dir outp." → 1700; "data dir" → 1701; "data dir" → 1703; "all lines low = all inp.")

DECIMAL	HEX	BINARY	$\overline{\text{HEX}}$
0	0	0000	F
1	1	0001	E
2	2	0010	D
3	3	0011	C
4	4	0100	B
5	5	0101	A
6	6	0110	9
7	7	0111	8
8	8	1000	7
9	9	1001	6
10	A	1010	5
11	B	1011	4
12	C	1100	3
13	D	1101	2
14	E	1110	1
15	F	1111	0

INTERVAL TIMER

WRT to @:	/N	INT
1704	1	D
1705	8	D
1706	64	D
1707	1024	D
170C	1	E
170D	8	E
170E	64	E
170F	1024	E

READ @		INT
1707	STATUS	
1706	COUNT	D
170E	COUNT	E

DISPLAY OUTPUT:

```
XX    XX    XX
00FB  00FA  00F9
```

JSR@ $1F1F

KEYBOARD INPUT:

JSR@ $1F6A

OUTPUT TO TTY:

char in A
JSR@ $1EA0

INPUT FROM TTY:

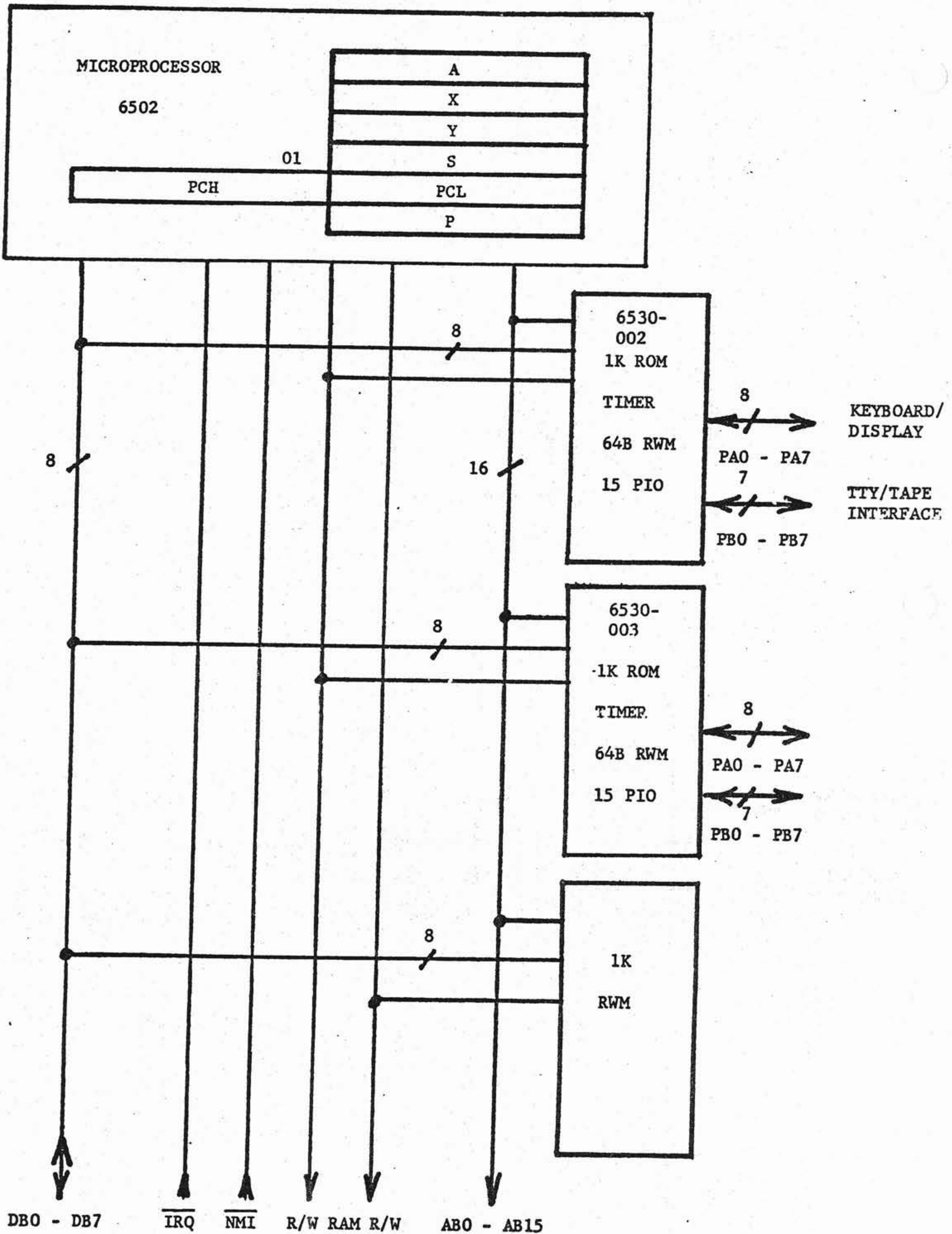
JSR@ $1E5A
char into A

ASCII/HEX CONVERSION TABLE

Char	7-Bit Hex No.	Char	7-Bit Hex No.	Char	7-Bit Hex No.	Char	7-Bit Hex No.
NUL	00						
SOH	01	!	21	A	41	a	61
STX	02	"	22	B	42	b	62
ETX	03	#	23	C	43	c	63
EOT	04	$	24	D	44	d	64
ENQ	05	%	25	E	45	e	65
ACK	06	&	26	F	46	f	66
BEL	07	'	27	G	47	g	67
BS	08	(	28	H	48	h	68
HT	09	)	29	I	49	i	69
LF	0A	*	2A	J	4A	j	6A
VT	0B	+	2B	K	4B	k	6B
FF	0C	,	2C	L	4C	l	6C
CR	0D	-	2D	M	4D	m	6D
SO	0E	.	2E	N	4E	n	6E
SI	0F	/	2F	O	4F	o	6F
DLE	10	0	30	P	50	p	70
DC1	11	1	31	Q	51	q	71
DC2	12	2	32	R	52	r	72
DC3	13	3	33	S	53	s	73
DC4	14	4	34	T	54	t	74
NAK	15	5	35	U	55	u	75
SYN	16	6	36	V	56	v	76
ETB	17	7	37	W	57	w	77
CAN	18	8	38	X	58	x	78
EM	19	9	39	Y	59	y	79
SUB	1A	:	3A	Z	5A	z	7A
ESC	1B	;	3B	[	5B		7B
FS	1C	<	3C	\	5C		7C
GS	1D	=	3D	]	5D	ALT	7D
RS	1E	>	3E	↑	5E	ESC	7E
US	1F	?	3F	←	5F	DEL	7F
SP	20	@	40	\	60	RUBOUT	

CHAR	DEFINITION	CHAR	DEFINITION
NUL	NULL	SO	SHIFT OUT
SOH	START OF HEADING, ALSO START OF MESSAGE	SI	SHIFT IN
STX	START OF TEXT; ALSO EOA, END OF ADDRESS	DLE	DATA LINK ESCAPE
ETX	END OF TEXT; ALSO EOM, END OF MESSAGE	DC1	DEVICE CONTROL 1
EOT	END OF TRANSMISSION (END)	DC2	DEVICE CONTROL 2
ENQ	ENQUIRY (ENQRY), ALSO WRU	DC3	DEVICE CONTROL 3
ACK	ACKNOWLEDGE, ALSO RU	DC4	DEVICE CONTROL 4
BEL	RINGS THE BELL	NAK	NEGATIVE ACKNOWLEDGE
BS	BACKSPACE	SYN	SYNCHRONOUS IDLE (SYNC)
HT	HORIZONTAL TAB	ETB	END OF TRANSMISSION BLOCK
LF	LINE FEED OR LINE SPACE (NEW LINE) ADVANCES PAPER TO NEXT LINE BEGINNING OF LINE	CAN	CANCEL (CANCL)
		EM	END OF MEDIUM
		SUB	SUBSTITUTE
		ESC	ESCAPE, PREFIX
		FS	FILE SEPARATOR
VT	VERTICAL TAB (VTAB)	GS	GROUP SEPARATOR
FF	FORM FEED TO TOP OF NEXT PAGE (PAGE)	RS	RECORD SEPARATOR
		US	UNIT SEPARATOR
CR	CARRIAGE RETURN	[illegible]	[illegible]

KIM BLOCK DIAGRAM



KIM INTERFACIN' DATA SHEET

APPLICATION CONNECTOR PLUG

FUNCTION	PIN #	FUNCTION	PIN #
PB0	1	PA7	16
PB1	2	PA6	15
PB2	3	PA5	14
PB3	4	PA4	13
PB4	5	PA3	12
PB5	6	PA2	11
INT(orange)	7	PA1	10
PB7*	8	PA0	9

*see p. H-7 for details about this line

EXPANSION CONNECTOR PLUG

FUNCTION	PIN #	FUNCTION	PIN #
AB0	1	DB7	16
AB1	2	DB6	15
AB2	3	DB5	14
AB3	4	DB4	13
K(green)	5	DB3	12
R/W̄	6	DB2	11
Ø2	7	DB1	10
RAM R/W	8	DB0	9

PROGRAMMABLE I/O LINES

PA DATA REGISTER 1700   PA DIRECTION REGISTER 1701 (0=input, 1=output)
PB DATA REGISTER 1702   PB DIRECTION REGISTER 1703

TAPE RECORDER CONNECTIONS

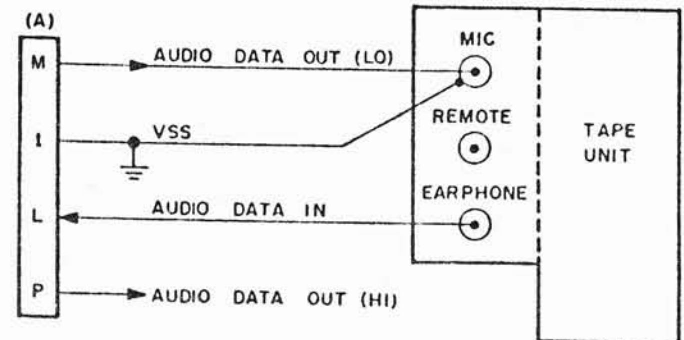
FUNCTION	PIN #	COLOR CODE	NOTES
GND	K	GREY	GND
AUDIO IN	L	BLUE	FROM EARPHONE OUTPUT
AUDIO OUT	M	RED	TO MIC INPUT

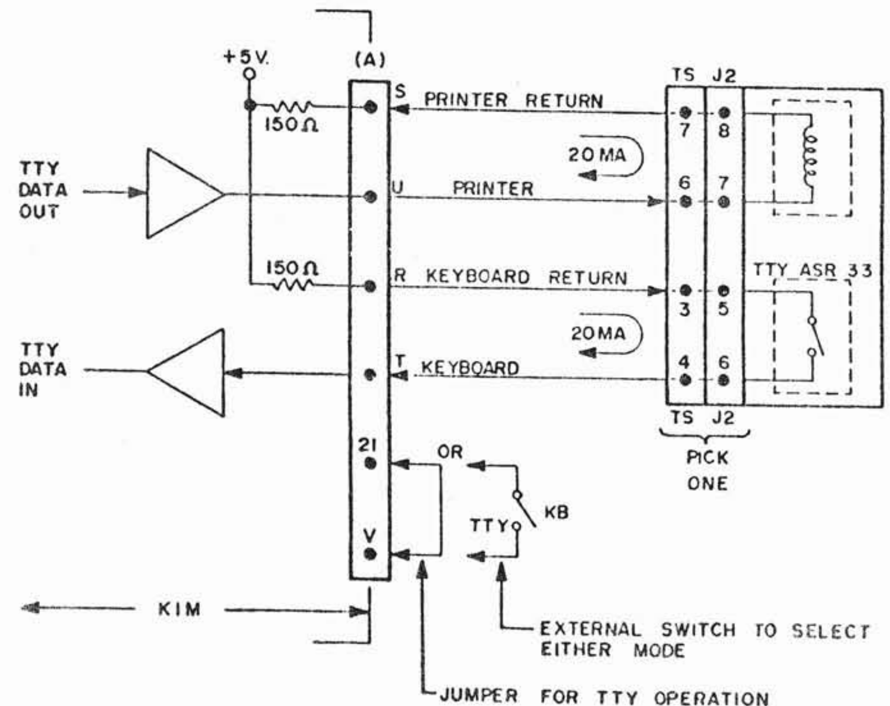
TTY/CRT CONNECTIONS

FUNCTION	PIN #	COLOR CODE	NOTES
KEYBD RET.	R	BROWN	(+)
PRINTER RET.	S	BROWN	(+)
KEYBOARD	T	VIOLET	20 mA current loop
PRINTER	U	YELLOW	20 mA current loop

TTY BAUD RATE CONTROL

BAUD	110	300	600	1200	2400	4800	9600
17F2	79	EA	75	38	18	0A	02
17F3	02	00	00	00	00	00	00





KIM MONITOR IMPORTANT ADDRESSES

00EF	USER PC LOW BYTE
00F0	USER PC HIGH BYTE
00F1	USER STATUS REGISTER
00F2	USER STACK POINTER
00F3	USER ACCUMULATOR
00F4	USER Y REGISTER
00F5	USER X REGISTER
00F6	CHECKSUM
00F7	CHECKSUM
00F9	STORAGE FOR RIGHTHAND DISPLAY DIGIT PAIR
00FA	STORAGE FOR CENTER DISPLAY DIGIT PAIR
00FB	STORAGE FOR LEFTHAND DISPLAY DIGIT PAIR
1700	PORT A DATA
1701	PORT A DIRECTION CONTROL REGISTER
1702	PORT B DATA
1703	PORT B DIRECTION CONTROL REGISTER
1704-1707	INTERVAL TIMER #1
170C-170F	INTERVAL TIMER #1
1744-1747	INTERVAL TIMER #2
174C-174F	INTERVAL TIMER #2
17F2-17F3	SERIAL I/O BAUD RATE CONTROL
17F5	TAPE DUMP STARTING ADDRESS LOW BYTE
17F6	TAPE DUMP STARTING ADDRESS HIGH BYTE
17F7	TAPE DUMP ENDING ADDRESS+1 LOW BYTE
17F8	TAPE DUMP ENDING ADDRESS+1 HIGH BYTE
17F9	TAPE FILE INDENTIFICATION NUMBER
17FA	NMI VECTOR LOW BYTE
17FB	NMI VECTOR HIGH BYTE
17FE	IRQ VECTOR LOW BYTE
17FF	IRQ VECTOR HIGH BYTE
1800	ENTRY POINT FOR TAPE DUMP ROUTINE
1873	ENTRY POINT FOR TAPE LOAD ROUTINE
1C00	NONDESTRUCTIVE MONITOR ENTRY POINT
1C4F	DESTRUCTIVE MONITOR ENTRY POINT
1EA0	SERIAL OUTPUT ENTRY POINT
1E5A	SERIAL INPUT ENTRY POINT
1F1F	ENTRY POINT FOR DIGIT DISPLAY ROUTINE
1F6A	ENTRY POINT FOR KEYBOARD READ ROUTINE

APPENDIX C

C. KIM SOFTWARE COLLECTION

Cass R. Lewart
Holmdel, N. J.

DISPLAY ROUTINE

This routine will display any program showing each successive location and the contents of that location. The routine is fully relocatable. By storing in the 17FA and 17FB locations the starting address of this routine one can use the ST key to start the program. The display can be stopped by pressing RS and continued by pressing ST again. The program starts displaying consecutive locations starting with the location shown in the display by pressing ST. The second program MULT controls the display time. With value 04 it is 0.4 sec per location.

```
00   A2 04      THREE   LDX#   MULT
02   8A         TWO     TXA
03   48                 PHA
04   A9 62              LDA#   $62      .1 sec/cycle
06   8D 47 17           STA@   $1747    Load timer
09   20 19 1F   ONE     JSR@   SCANDS   Display
0C   2C 47 17           BIT@   $1747    Check timer
0F   10 F8              BPLr   ONE
11   68                 PLA
12   AA                 TAX
13   CA                 DEX
14   D0 EC              BNEr   TWO
16   E6 FA              INCz   $FA
18   D0 E6              BNEr   THREE
1A   E6 FB              INCz   $FB
1C   D0 E2              BNEr   TWO
```

e.g. to start displaying at 210: AD,0,2,1,0,ST
if the DISPLAY starts at 300: AD, 1,7,F,A,DA,0,0,+,0,3,AD,go to desired location,ST,....

1747 loc of interval timer

Jim Butterfield
Toronto

DIRECTORY: A KIM-1 UTILITY PROGRAM

Program DIRECTORY allows you 254 program IDs to choose from ... enough for most program libraries with some to spare. The program is fully relocatable, so put it anywhere convenient. Start at the first instruction (0000 in the listing). Incidentally, 0001 to 001D of this program are functionally identical to the KIM monitor 188C to 18C1.

After you start the program, start your audio tape input. When DIRECTORY finds a program, it will display the Start Address (first four digits) and the Program ID. Hit any key and it will scan for the next program.

```
0000 D8        GO    CLD
0001 A9 07           LDA#   $07        Directional reg
0003 8D 42 17        STA@   SBD
0006 20 41 1A  SYN   JSR@   RDBIT      Scan thru bits...
0009 46 F9           LSRz   INH        ..shifting new bit
000B 05 F9           ORAz   INH        ..into left of
000D 85 F9           STAz   INH        ..byte INH
000F C9 16     TST   CMP#   $16        SYNC character?
0011 D0 F3           BNEr   SYN        no, back to bits
0013 20 24 1A        JSR@   RDCHT      get a character
0016 C6 F9           DECz   INH        count 22 SYNC's
0018 10 F5           BPLr   TST
001A C9 2A           CMP#   $2A        then test astk
001C D0 F1           BNEr   TST        ..or SYNC
001E A2 FD           LDX#   $FD        if asterisk,
0020 20 F3 19  RD    JSR@   RDBYT      stack 3 bytes
0023 95 FC           STAzx  POINTH+1   into display
0025 E8              INX               area
0026 30 F8           BMIr   RD
0028 20 1F 1F  SHOW  JSR@   SCANDS     ...and shine
002B D0 D3           BNEr   GO         until keyed
002D F0 F9           BEQr   SHOW       at's all folks
```

VU TAPE

Jim Butterfield
Toronto

Program VUTAPE lets you actually see the contents of a KIM format tape as it's going by. It shows the data going by very quickly, because of the tape speed .. but you can at least 'sense' the kind of material on the tape. In case of tape troubles, this should give you a hint as to the area of your problem: nothing? noise? dropouts? And you can prepare a test tape (see below) to check out the tape quality and your recorder. The test tape will also help you establish the best settings for your volume and tone controls.

Perhaps VUTAPE's most useful function, though, is to give you a 'feeling' for how data is stored on tape. You can actually watch the processor trying to synchronize into the bit stream. Once it's synched, you'll see the characters rolling off the tape ... until an END or illegal character drops you back into the sync mode again. It's educational to watch. And since the program is fairly short, you should be able to trace out just how the processor tracks the input tape.

VUTAPE starts at location 0000 and is fully relocatable (so you can load it anyplace it fits).

```
0000 D8          START   CLD
0001 A9 7F               LDA#    $7F
0003 8D 41 17            STA@    PADD     set display dir reg
0006 A9 13       SYN     LDA#    $13      ..window 6 and tape in
0008 85 E0               STAz    POINT      and keep pointer
000A 8D 42 17            STA@    SBD
000D 20 41 1A            JSR@    RDBIT      get a bit and
0010 46 F9               LSRz    INH        ..slip it into
0012 05 F9               ORAz    INH        ..the right-hand
0014 85 F9               STAz    INH        ..side;
0016 8D 40 17            STA@    SAD      show bit flag on display
0019 C9 16       TST     CMP#    $16       ..is it a SYNC?
001B D0 E9               BNEr    SYN      nope, keep 'em rolling
001D 20 24 1A            JSR@    RDCHT     yup, start grabbing...
0020 C9 2A               CMP#    $2A       .8 bits at a time and..
0022 D0 F5               BNEr    TST       ..if it's not an '*'..
0024 A9 00       STREAM  LDA#    $00       ..then start showing
0026 8D E9 17            STA@    SAVX      ..characters 1 at a time
0029 20 24 1A            JSR@    RDCHT
002C 20 00 1A            JSR@    PACKT     ..converting to hexadec..
002F D0 D5               BNEr    SYN       ..if legal
0031 A6 E0               LDXz    POINT
0033 E8                  INX
0034 E8                  INX               Move along to next..
0035 E0 15               CPX#    $15       ..display position
0037 D0 02               BNEr    OVER      (if last digit,..
0039 A2 09               LDX#    $09       ..reset to first)
003B 86 E0       OVER    STXz    POINT
003D 8E 42 17            STX@    SBD
0040 AA                  TAX               change character read
0041 BD E7 1F            LDA@x   TABLE       ..to segments and..
0044 8D 40 17            STA@    SAD         send to the display
0047 D0 DB               BNEr    STREAM      unconditional jump
```

PROGRAM TO CHECK OUT TAPES/RECORDERS

Make a test tape containing an endless stream of SYNC characters with the following program:

```
0000 A0 BF      GO      LDY#    $BF         directional..
0002 8C 43 17           STY@    PBDD            ..registers
0005 A9 16      LP      LDA#    $16           SYNC
0007 20 7A 19           JSR@    OUTCH         ..out to tape
000A D0 F9              BNEr    LP
```

Now use program VUTAPE. The display should show a steady synchronization pattern. Try playing with your controls and see over what range the pattern stays locked in. The wider the range, the better your cassette/ recorder.

Jim Butterfield
Toronto

SUPERTAPE

```
0100 A9 AD      DUMPT   LDA#  $AD      op code LDA
0102 8D EC 17           STA@  VEB
0105 20 32 19           JSR@  INTVEB   set up subrtn
0108 A9 27              LDA#  $27
010A 85 E1              STAz  GANG     flag to go to SBD
010C A9 BF              LDA#  $BF
010E 8D 43 17           STA@  PBDD     open the channels
0111 A2 64              LDX#  $64       send 100...
0113 A9 16              LDA#  $16         ..SYNC chars
0115 20 61 01           JSR@  HIC
0118 A9 2A              LDA#  $2A       send asterisk
011A 20 88 01           JSR@  OUTCHT
011D AD F9 17           LDA@  ID        then the ID
0120 20 70 01           JSR@  OUTBIT
0123 AD F5 17           LDA@  SAL       followed by
0126 20 6D 01           JSR@  OUTBTC     the start address
0129 AD F6 17           LDA@  SAH         (low and high)
012C 20 6D 01           JSR@  OUTBTC
012F 20 EC 17   DUMPT4  JSR@  VEB       get memory word
0132 20 6D 01           JSR@  OUTBTC     and send it
0135 20 EA 19           JSR@  INCVEB    on to next address
0138 AD ED 17           LDA@  VEB+1
013B CD F7 17           CMP@  EAL       is the address..
013E AD EE 17           LDA@  VEB+2      ..at the end?
0141 ED F8 17           SBC@  EAH
0144 90 E9              BCCr  DUMPT4    no, go back;
0146 A9 2F              LDA#  $2F       yes, send end-data
0148 20 88 01           JSR@  OUTCHT
014B AD E7 17           LDA@  CHKL      ..and checksum
014E 20 70 01           JSR@  OUTBT
0151 AD E8 17           LDA@  CHKH      ..hi and low..
0154 20 70 01           JSR@  OUTBT
0157 A2 02              LDX#  $02       send two..
0159 A9 04              LDA#  $04       EOT characters
015B 20 61 01           JSR@  HIC
015E 4C 5C 18           JMP@  DISPZ     and we're finished
                ;  subroutines follow here
0161 86 E0      HIC     STXz  TIC       count
0163 48         HIC1    PHA
0164 20 88 01           JSR@  OUTCHT    send character
0167 68                 PLA             ..and bring it back
0168 C6 E0              DECz  TIC
016A D0 F7              BNEr  HIC1      do it again
016C 60                 RTS
```

```
016D 20 4C 19  OUTBTC  JSR@   CHKT       compute checksum
0170 48        OUTBT   PHA               save the character
0171 4A                LSRa
0172 4A                LSRa              ..and take its
0173 4A                LSRa               four left bits..
0174 4A                LSRa
0175 20 7D 01          JSR@   HEXOUT      write 'em ...
0178 68                PLA               now the 4 right bits..
0179 20 7D 01          JSR@   HEXOUT
017C 60                RTS

017D 29 0F     HEXOUT  AND#   $0F        remove unwanted bits
017F C9 0A             CMP#   $0A         change to ASCII by..
0181 18                CLC                   adding:
0182 30 02             BMIr   HEX1
0184 69 07             ADC#   $07         $37 if A to F
0186 69 30     HEX1    ADC#   $30         $30 if numeric
0188 A0 08     OUTCHT  LDY#   $08         for the eight bits..
018A 84 E2             STYz   COUNT
018C A0 02     TRY     LDY#   $02         send 3 units
018E 84 E3             STYz   TRIB        starting at 3600 hertz
0190 BE BE 01  ZON     LDX@y  NPUL         number of half cycles
0193 48                PHA                 keep the character
0194 2C 47 17  ZON1    BIT@   CLKRDI       wait for the previous..
0197 10 FB             BPLr   ZON1            cycle to complete
0199 B9 BF 01          LDAay  TIMG         get the time to the..
019C 8D 44 17          STA@   CLK1T          ..next pulse ($7E or C3)
019F A5 E1             LDAz   GANG
01A1 49 80             EOR#   $80          flip between 1 and 0
01A3 8D 42 17          STA@   SBD
01A6 85 E1             STAz   GANG
01A8 CA                DEX                have we sent all the cycles?
01A9 D0 E9             BNEr   ZON1        nope, send another one
01AB 68                PLA                get back the character
01AC C6 E3             DECz   TRIB        one less unit to send
01AE F0 05             BEQr   SETZ        and the last one's here
01B0 30 07             BMIr   ROUT        none left?  quit
01B2 4A                LSRa               take next bit
01B3 90 DB             BCCr   ZON         ..and if it's a one..
01B5 A0 00     SETZ    LDY#   $00         switch to 2400 cycles/sec
01B7 F0 D7             BEQr   ZON        unconditional return
01B9 C6 E2     ROUT    DECz   COUNT        one less bit
01BB D0 CF             BNEr   TRY         any more? go back
01BD 60                RTS
               ; frequency/density controls
01BE 02        NPUL    .BYTE  $02         two pulses; one cycle!
01BF C3 03 7E  TIMG    .BYTE  $C3,$03,$7E
                       end
```

Speed	Timing Data:	$01BE	$01C0
X3		04	06
X6		02	03

TAPE DUPLICATION PROGRAM

Jim Butterfield

```
1780 A9 27      START   LDA#   $27        SBD value
1782 A2 3F      GO      LDX#   $3F        set directional register to
1784 8E 43 17           STX@   PBDD       input
1787 A2 07              LDX#   $07        PB5 (cont) set for input
1789 8E 42 17           STX@   SBD
178C A0 5E              LDY#   94         high frequency
178E 2C 42 17           BIT@   SBD        zero or one?
1791 10 02              BPLr   OVER
1793 A0 A3              LDY#   163        low frequency
1795 A2 BF      OVER    LDX#   $BF        set directional register to
1797 8E 43 17           STX@   PBDD       output
179A 49 80              EOR#   $80        reverse output bit
179C 8D 42 17           STA@   SBD        and send it
179F 8C 44 17           STY@   CLKIT      set timer
17A2 2C 47 17   WAIT    BIT@   CLKSTAT    and wait
17A5 10 FB              BPLr   WAIT
17A7 30 D9              BMIr   GO
```

Connect your two cassette recorders in the usual way at the AUDIO IN and AUDIO OUT points. With the program running, start the recorders. All programs will be copied from one tape to the other. This program works on speeds up to 3X. If bad copies are obtained, try reducing the volume on the playback machine.

MOVE-A-BLOCK

Edward J. Bechtel, M.D.
Newport Beach, Calif.

The MOVE-A-BLOCK program will move a block of bytes up to 256 bytes long forewards or backwards any distance. The block can be across page boundaries -- it does not have to reside in one page. The starting address and ending address of the block is entered in 00E0 - 00E3. The NEW starting address of the moved block (i.e., where you want to move it) is entered at 00E4 - 00E5. I located it in 1780 to be generally out of the way, but if you wish, you can use it to relocate itself anywhere.

The program calculates whether the move is forewards or backwards, then moves from the top up, or from the bottom down. The number of spaces the block is moved (in signed notation) is stored by the program in 00E6 - 00E7, and the number of bytes that were moved is stored in 00E8. Also, the new ending address of the moved block is automatically placed in 00E2 - 00E3, for subsequent use.

```
1780 38                  SEC
1781 A5 E4               LDAz    $E4
1783 E5 E0               SBCz    $E0
1785 85 E6               STAz    $E6
1787 A5 E5               LDAz    $E5
1798 E5 E1               SBCz    $E1
178B 85 E7               STAz    $E7
178D 90 18               BCCr    MOVEB
178F 38         MOVEF    SEC
1790 A5 E2               LDAz    $E2
1792 E5 E0               SBCz    $E0
1794 A8                  TAY
1795 84 E8               STYz    $E8
1797 E6 E8               INCz    $E8
1799 B1 E0      LOOP1    LDAiy   $E0
179B 91 E4               STAiy   $E4
179D 88                  DEY
179E D0 F9               BNEr    LOOP1
17A0 B1 E0               LDAiy   $E0
17A2 91 E4               STAiy   $E4
17A4 88                  DEY
17A5 30 14               BMIr    END
17A7 38         MOVEB    SEC
17A8 A5 E2               LDAz    $E2
17AA E5 E0               SBCz    $E0
17AC 85 E8               STAz    $E8
17AE E6 E8               INCz    $E8
17B0 A0 00               LDY#    $00
17B2 B1 E0      LOOP2    LDAiy   $E0
17B4 91 E4               STAiy   $E4
17B6 C8                  INY
17B7 C4 E8               CPYz    $E8
17B9 D0 F7               BNEr    LOOP2
17BB 18         END      CLC
17BC A5 E2               LDAz    $E2
17BE 65 E6               ADCz    $E6
17C0 85 E2               STAz    $E2
17C2 A5 E3               LDAz    $E3
17C4 65 E7               ADCz    $E7
17C6 85 E3               STAz    $E3
17C8 4C 4F 1C            JMP@    START
```

```
00E0 = SAL)
00E1 = SAH) Original
          ) block of
00E2 = EAL) bytes
00E3 = EAH)

00E4 = SAL) New location
00E5 = SAH)

00E6 = dif L) Number of spaces
00E7 = dif H) block is moved
              (signed notation)
00E8 = Number of bytes in block
```

HEX DUMP

by J.B. Ross

Here is a program to print out machine language programs in hexadecimal format. To use the program, load the starting address of the dump in $17F5 (SAL) and $17F6 (SAH), the ending address +1 in $17F7 (EAL+1) and $17F8 (EAH), then run HEX DUMP starting at $0100. HEX DUMP is relocateable so you can move it to other memory locations as needed. As written, HEX DUMP centers the print-out on an 80 character line with 11 spaces on the left. The print-out itself requires 53 spaces. To modify the left margin, change the data in locations $0113 and $0137.

HEX DUMP

```
100 AD F5 17  START  LDA@    $17F5    get low starting address
103 85 FA            STAZ    POINTL   save it in POINTL
105 AD F6 17         LDA@    $17F6    get high starting address
108 85 FB            STAZ    POINTH   save it in POINTH
10A 20 2F 1E         JSR@    CRLF     print CR/LF
10D A9 0A            LDA#    'LF'     print another LF
10F 20 A0 1E         JSR@    OUTCH
112 A2 0F            LDX#    $0F      print 15 spaces on left
114 20 9E 1E  LOOP1  JSR@    OUTSP
117 CA               DEX
118 D0 FA            BNER    LOOP1
11A A2 10            LDX#    $10      print heading:
11C A9 FF            LDA#    $FF      start with A at -1
11E 48               PHA              save A
11F 20 9E 1E         JSR@    OUTSP    print 1 space
122 20 9E 1E  LOOP2  JSR@    OUTSP    print 1 space
125 68               PLA              restore A
126 18               CLC
127 69 01            ADC#    $01      add 1 to A
129 48               PHA              save A
12A 20 3B 1E         JSR@    PRTBYT   print A as hex number
12D CA               DEX
12E D0 F2            BNER    LOOP2
130 20 2F 1E         JSR@    CRLF     print CR/LF
133 20 2F 1E  LOOP5  JSR@    CRLF     print CR/LF
136 A2 0B            LDX#    $0B      print 11 spaces on left
138 20 9E 1E  LOOP3  JSR@    OUTSP
13B CA               DEX
13C D0 FA            BNER    LOOP3
13E A2 10            LDX#    $10      set up data counter
140 20 1E 1E         JSR@    PRTPNT   print address
143 20 9E 1E         JSR@    OUTSP    space
146 20 9E 1E  LOOP4  JSR@    OUTSP    space
149 A0 00            LDY#    $00      zero Y
14B B1 FA            LDAIY   POINTL   get data from address
14D 20 3B 1E         JSR@    PRTBYT   print data
150 20 63 1F         JSR@    INCPT    increment address pointer
153 A5 FB            LDAZ    POINTH   test for maximum address
155 CD F8 17         CMP@    $17F8
158 90 09            BCCR    MORE
15A A5 FA            LDAZ    POINTL
15C CD F7 17         CMP@    $17F7
15F 90 02            BCCR    MORE
161 B0 06            BCSR    DONE
163 CA        MORE   DEX              decrement data counter
164 D0 E0            BNER    LOOP4    repeat if counter not zero
166 18               CLC              go to LOOP5
167 90 CA            BCCR    LOOP5
169 20 2F 1E  DONE   JSR@    CRLF     print two blank lines
16C 20 2F 1E         JSR@    CRLF
16F 4C 4F 1C         JMP@    KIM      return to monitor
```

Joe Laughter
Memphis, Tenn.

FREQUENCY COUNTER ROUTINE

This routine counts frequency using input PB0 at a maximum rate of 20 KHz. It counts DATA for 1 second. To count for 10 seconds load $29 into address 60. It uses PB7 for int. req. (connect PB7 to IRQ.).

```
0000 A9 01              LDA#   $01
0002 85 65              STAz   TMECNT
0004 F8                 SED
0005 A9 36              LDA#   INTLOW    set int. vector
0007 8D FE 17           STA@   $17FE
000A A9 00              LDA#   INTHIGH
000C 8D FF 17           STA@   $17FF
000F 58                 CLI
0010 00                 BRK
0011 EA                 NOP
0012 AD 02 17  CKLOW    LDA@   PB        check for input low
0015 29 01              AND#   $01
0017 D0 F9              BNEr   CKLOW
0019 AD 02 17  CKHIGH   LDA@   PB        check for input high
001C 29 01              AND#   $01
001E F0 F9              BEQr   CKHIGH
0020 18                 CLC              add count to total
0021 A9 01              LDA#   $01
0023 65 F9              ADCz   $F9
0025 85 F9              STAz   $F9
0027 A9 00              LDA#   $00
0029 65 FA              ADCz   $FA
002B 85 FA              STAz   $FA
002D A9 00              LDA#   $00
002F 65 FB              ADCz   $FB
0031 85 FB              STAz   $FB
0033 4C 12 00           JMP@   CKLOW
0036 48        INT      PHA              check time
0037 A9 90              LDA#   $90
0039 8D 04 17           STA@   $1704
003C 2C 07 17           BIT@   $1704
003F 10 FB              BPLr   DELAY
0041 A9 F4              LDA#   $F4       set timer for another int.
0043 8D 0F 17           STA@   $170F
0046 C6 65              DECz   TMECNT    check remaining time
0048 F0 02              BEQr   DISP      if zero display counts
004A 68                 PLA
004B 40                 RTI
004C A9 FF     DISP     LDA#   $FF       set display loop count
004E 85 66              STAz   SCANCT
0050 20 1F 1F  OUT      JSR@   SCANDS    output data
0053 C6 66              DECz   SCANCT    dec. loop count
0055 D0 F9              BNEr   OUT       rept. display till loop
0057 A9 00              LDA#   $00       count is zero
```

```
0059 85 F9          STAz   $F9      set total counts to zero
005B 85 FA          STAz   $FA
005D 85 FB          STAz   $FB
005F A9 05          LDA#   $05      reset 1 sec timer
0061 85 65          STAz   TMECNT
0063 68             PLA
0064 40             RTI
0065 05             *DATA  (TMECNT)
0066 FF             *DATA  (SCANCT)
```

ANALOG TO DIGITAL CONVERSION DEMONSTRATION PROGRAM

Display ADC Output in HEX Format

```
0000 A9 FF     START   LDA#    $FF       set PA port to output
0002 8D 01 17          STA@    $1701
0005 AD 03 17          LDA@    $1703     set PB4 to be input
0008 29 EF             AND#    $EF
000A 8D 03 17          STA@    $1703
000D 20 80 00  LOOP    JSR@    ADC       call ADC subroutine
0010 85 F9             STAz    $F9       store ADC output in right display
0012 20 1F 1F          JSR@    SCANDS    display data
0015 4C 0D 00          JMP@    LOOP      loop back for more data
```

Display ADC Output in BCD Format

```
0020 A9 FF     START   LDA#    $FF       set PA port to output
0022 8D 01 17          STA@    PADD
0025 AD 03 17          LDA@    PBDD      set PB4 to be input
0028 29 EF             AND#    $EF
002A 8D 03 17          STA@    PBDD
002D 20 80 00  READ    JSR@    ADC       read ADC
0030 85 E7             STAz    HEDEC-L   set up data for binary to BCD conversion
0032 A2 00             LDX#    $00
0034 86 E6             STXz    HEDEC-H
0036 20 00 02          JSR@    HEDEC     call binary to BCD conversion routine
0039 A6 E1             LDXz    $E1       get BCD result high
003B 86 FB             STXz    $FB       store result in left display
003D A6 E2             LDXz    $E2       get BCD result low
003F 86 FA             STXz    $FA       store result in middle display
0041 A2 00             LDX#    $00       zero the right display
0043 86 F9             STXz    $F9
0045 20 1F 1F          JSR@    SCANDS    display final BCD value
0048 4C 2D 00          JMP@    READ      loop back for more data
```

note: In order to perform the binary to BCD conversion, you must load the HEDEC program into the memory starting at address $0200.

REAL-TIME CLOCK

KIM-1 User Notes v.1 #4
Charles H. Parsons
80 Longview Rd.
Monroe, CT 06468

This program utilizes the interval timer to produce an NMI interrupt every 249,856 microseconds. A fine adjustment to 1/4 second is done with the same time in the interuppt program. This fine adjustment can be varied by changing the number in location $03AB. A display routine is included which shows the time on the KIM-1 display. You can exit this routine and get back to the monitor by pressing the "1" key.

To run the clock program you must connect PB7 to expansion connector pin 6 and set up the NMI interrupt vector by storing $A5 in $17FA and $03 in $17FB. The clock is set by using the KIM monitor to enter the current time into the HR, MIN, and SEC locations given below.

```
1/4 SEC = $0080   1/4 second counter
    SEC = $0081   second counter
    MIN = $0082   minute counter
     HR = $0083   hour counter
1/2 DAY = $0084   day counter for am-pm
```

Run the display program once starting at $0370 to get the interrupt routine going, then re-enter the display routine at $0379 whenever **you want to show the time.**

REAL-TIME CLOCK - DISPLAY ROUTINE

```
0370 A9 00     START  LDA#  $00     zero 1/4 second memory
0372 85 80            STAZ  QSEC
0374 A9 F4            LDA#  $F4     set timer to interrupt in 1/4 sec.
0376 8D 0F 17         STA@  TIMEF
0379 A5 81     DSPLY  LDAZ  SEC     get seconds
037B 85 F9            STAZ  $F9     send to right display pair
037D A5 82            LDAZ  MIN     get minutes
037F 85 FA            STAZ  $FA     send to middle display pair
0381 A5 83            LDAZ  HR      get hours
0383 85 FB            STAZ  $FB     sent to left display pair
0385 20 6A 1F         JSR@  GETKEY  check for "1" key pressed
0388 C9 01            CMP#  $01
038A D0 0D            BNER  ENDR
038C 20 1F 1F         JSR@  SCANDS  display time and delay
038F 20 6A 1F         JSR@  GETKEY  check for "1" key pressed again
0392 C9 01            CMP#  $01
0394 D0 03            BNER  ENDR
0396 4C 4F 1C         JMP@  MONTR   jump back to monitor if "1" pressed
0399 20 1F 1F         JSR@  SCANDS  display time again
039C 18               CLC           jump back to DSPLY to continue
039D 90 DA            BCCR  DSPLY
```

REAL-TIME CLOCK - INTERRUPT ROUTINE

```
03A5 48        RTCLK  PHA           save A
03A6 8A               TXA
03A7 48               PHA           save X
03A8 98               TYA
03A9 48               PHA           save Y
03AA A9 83            LDA#  $83     fine adjust timing
03AC 8D 04 17         STA@  TIME4
03AF 2C 07 17  TM     BIT@  TIMES   test timer status
03B2 10 FB            BPLR  TM      loop until time out
03B4 E6 80            INCZ  QSEC    count 1/4 seconds
03B6 A9 04            LDA#  $04     do four times before updating seconds
03B8 C5 80            CMPZ  QSEC
03BA D0 38            BNER  RTN
03BC A9 00            LDA#  $00     zero QSEC and update clock
03BE 85 80            STAZ  QSEC
03C0 18               CLC
03C1 F8               SED           change to decimal mode
03C2 A5 81            LDAZ  SEC     increment seconds
03C4 69 01            ADC#  $01
03C6 85 81            STAZ  SEC
03C8 C9 60            CMP#  60      until seconds = 60
03CA D0 28            BNER  RTN
03CC A9 00            LDA#  00      reset seconds to 00
03CE 85 81            STAZ  SEC
03D0 A5 82            LDAZ  MIN     increment minutes
03D2 18               CLC
03D3 69 01            ADC#  01
03D5 85 82            STAZ  MIN
03D7 C9 60            CMP#  60      until minutes = 60
03D9 D0 19            BNER  RTN
03DB A9 00            LDA#  00      reset minutes to 00
03DD 85 82            STAZ  MIN
03DF A5 83            LDAZ  HR      increment hours
03E1 18               CLC
03E2 69 01            ADC#  01
03E4 85 83            STAZ  HR
03E6 C9 12            CMP#  12      until hours = 12
03E8 D0 02            BNER  TH
03EA E6 84            INCZ  DAY     increment 1/2 day
03EC C9 13     TH     CMP#  13      check for 13 hours
03EE D0 04            BNER  RTN
03F0 A9 01            LDA#  01      start again with one
03F2 85 83            STAZ  HR
03F4 D8        RTN    CLD           return to binary mode
03F5 A9 FA            LDA#  $FA     set timer to interrupt in 249,856 sec
03F7 8D 0F 17         STA@  TIMEF
03FA 68               PLA           restore Y
03FB A8               TAY
03FC 68               PLA           restore X
03FD AA               TAX
03FE 68               PLA           restore A
03FF 40               RTI           return from interrupt
```

TIMER (STOPWATCH)

Kim-1 User Notes
v. 1 #2

Joel Swank #186
4655 S. W. 142nd
Beaverton, OR 97005

TIMER turns KIM-1 into a digital stopwatch showing up to 99 minutes and 59.99 seconds. It is designed to be accurate to 50 microseconds per second. The KIM-1 interval timer is used to count 9984 machine cycles and the instructions between time-out and the reset of the timer make up the remaining 16 cycles needed to produce a time delay of 0.0100 sec. The keyboard controls the routine as follows:

KEY	FUNCTION
0	stop
1	start
2	reset
3	print time on terminal
4	return to KIM monitor

STOPWATCH

```
0300 A9 79     BAUDR   LDA#  $79      set baud rate to 110 for printer
0302 8D F2 17          STA@  $17F2
0305 A9 02             LDA#  $02
0307 8D F3 17          STA@  $17F3
030A A9 00     RESET   LDA#  $00      zero display
030C 85 F9             STAZ  INH
030E 85 FA             STAZ  POINTL
0310 85 FB             STAZ  POINTH
0312 20 1F 1F  HOLD    JSR@  SCANDS   light display
0315 20 6A 1F          JSR@  GETKEY   read keyboard
0318 C9 04             CMP#  $04      key 4
031A D0 03             BNER  NOQUIT
031C 4C 64 1C          JMP@  CLEAR    return to KIM monitor
031F C9 03     NOQUIT  CMP#  $03      key 3
0321 D0 1F             BNER  NOPRT
0323 A5 FB             LDAZ  POINTH
0325 20 3B 1E          JSR@  PRTBYT   print time on terminal
0328 A9 3A             LDA#  ' : '
032A 20 A0 1E          JSR@  OUTCH
032D A5 FA             LDAZ  POINTL
032F 20 3B 1E          JSR@  PRTBYT
0332 A9 2E             LDA#  ' . '
0334 20 A0 1E          JSR@  OUTCH
0337 A5 F9             LDAZ  INH
0339 20 3B 1E          JSR@  PRTBYT
033C 20 2F 1E          JSR@  CRLF     end of print routine
033F 38                SEC            jump to HOLD
0340 B0 D0             BCS   HOLD
0342 C9 02     NOPRT   CMP#  $02      key 2
0344 F0 C4             BEQR  RESET    back to zero
0346 C9 01             CMP#  $01      key 1
0348 D0 C8             BNER  HOLD
034A A9 9C             LDA#  $9C
034C 8D 06 17          STA@  TIMSET   set timer
034F 20 1F 1F  DISPL   JSR@  SCANDS   display value
0352 AD 07 17  EXPCK   LDA@  TIMGET   check timer
0355 F0 FB             BEQR  EXPCK    wait loop
0357 8D 00 1C          STA@  ROM      delay 4 usec.
035A A9 9C             LDA#  $9C      set timer
035C 8D 06 17          STA@  TIMSET
035F 18                CLC            set flags
0360 F8                SED            decimal mode
0361 A5 F9             LDAZ  INH
0363 69 01             ADC#  $01      increment hundredths
0365 85 F9             STAZ  INH
0367 A5 FA             LDAZ  POINTL
0369 69 00             ADC#  $00      increment seconds
036B 85 FA             STAZ  POINTL
036D C9 60             CMP#  $60      stop at 60
036F D0 0B             BNER  CKEY
0371 A9 00             LDA#  $00
0373 85 FA             STAZ  POINTL   zero seconds
0375 A5 FB             LDAZ  POINTH
0377 18                CLC
0378 69 01             ADC#  $01      increment minutes
037A 85 FB             STAZ  POINTH
037C D8        CKEY    CLD
037D 20 6A 1F          JSR@  GETKEY   read keyboard
0380 C9 00             CMP#  $00      key 0
0382 D0 CB             BNER  DISPL
0384 F0 8C             BEQR  HOLD     stop
```

H. T. Gordon
Berkeley, Calif.

HEDEC

HEDEC converts a 4-digit hex number in 00 E6 (hi byte) and 00 E7 (lo byte) into a decimal equivalent stored in 00 E0, 00 E1, and 00 E2. It uses 00 E3, 00 E4, and 00 E5 to store calculated conversion factors for each of 16 binary bits. Length: 67 bytes. Conversion times: 0.7 millisec for hex 0000, 1.5 ms for hex 1111, 1.4 ms for hex 8080, and 2.12 ms for hex FFFF. Times are proportional to the number of binary 1 bits, not to the numerical value.

```
0200 F8        (sets decimal mode)
     98        (pushes Y, then X index into stack)
     48
     8A
     48
0205 A9 00     (zeros 00 E0 to 00 E5 in a loop)
     A2 06     (sets X-index for 6 operations)
     95 DF     (zero-page, X storing)
     CA
020C D0 FB
     E6 E5     (increments 00 E5 to 01, to be first conversion factor)
0210 A5 E7     (accumulator pick-up of lo hex byte)
0212 48        (stored in stack)
     A0 08     (sets Y-index for testing of 8 bits)
0215 68        (pulls hex byte from stack)
     4A        (one logical shift right, lowest bit in carry)
     48        (stores shifted hex byte in stack)
0218 90 0C     (if carry clear, bit was a zero. skip to 0226)
     A2 03     (if not, do triple-precision add of conversion factor
     18        to the decimal locations)
021D B5 E2
     75 DF
     95 DF
     CA
0224 D0 F7
0226 A2 03     (next conversion factor always calculated, doubling
     18        previous factor by adding it to itself, giving a
     B5 E2     sequence 1, 2, 4, 8, .... to final 65536 (not used))
     75 E2
     95 E2
     CA
0230 D0 F7
     88        (DEY)
0233 D0 E0     (if not zero, back to 0215 for next bit)
0235 68        (this PLA stack pull needed to equalize PHAs and PLAs)
     A5 E3     (LDA highest conversion factor location)
0238 D0 04     (if not zero, job is finished, so exit)
     A5 E6     (if zero, load hi hex byte)
023C D0 D4     (if not zero, back to 0212 for bit testing)
023E 68        (restore X, then Y, indexes)
     AA
     68
     A8
0242 D8        (clear decimal mode)
0243 60        (RTS)
```

BINARY MATH ROUTINES

KIM-1 User Notes v1 #3
H. T. Gordon
Berkeley, Calif

MULTIA SUBROUTINE

Program MULTIA (second, revised version) does binary multiplication of two 8-bit numbers that have been stored (before the JSR to MULTIA) in 00E3 and 00E4 and are destroyed by the operation of the subroutine. The hi 8 bits of the product are stored in 00E0 and the low 8 bits in 00E1; the subroutine initially zeros these locations, and also 00E2. Operations use LSRs on the multiplier in 00E4 to move up to 8 bits in sequence into the carry flag. If the carry is set, the multiplicand (in 00E2 and 00E3) is double-precision added to the product locations. If bits remain in the multiplier (00E4 not zero), the multiplicand is shifted left in the 16 bits of 00E2-00E3; otherwise the subroutine exits. Program length: 36 bytes. Maximum product (FF X FF) is FE01 or decimal 65025, with execution time about 380 microseconds. Time declines to 240 microseconds for 80 X 80. 160 microseconds for 10 X 10, 70 microseconds for 01 X 01, 40 microseconds for 00 X 00.

```
000A A9 00     (zeros locations 00E0 to 00E2)
     85 E2
     85 E1
     85 E0
0012 46 E4     (LSR 00E4, lowest bit into carry)
0014 90 0D     (if carry clear, skip the addition, go to 0023)
0016 18        (CLC starts double-precision add)
     A5 E1
     65 E3     (running totals stored in 00E0-00E1)
     85 E1
001D A5 E0
     65 E2
     85 E0
0023 A5 E4     (LDA of 00E4, zero flag set if zero)
     F0 06     (exit to 002D if zero)
0027 06 E3     (ASL shifts highest bit of 00E3 into carry,
     26 E2     ROL shifts carry into lowest bit of 00E2)
002B 90 E5     (carry is always clear, so back to 0012)
002D 60        (RTS exit)
```

NOTE: This subroutine assumes that the processor is in the binary (not the decimal mode)! It should not be necessary for subroutines to protect themselves (by a CLD) from this problem.

H. T. Gordon
Berkeley, Calif.

SUBROUTINE DIVIDA

This software gives the quotient, to 16-bit or better precision, from division of any hex number from 0001 to FFFF by any hex number from 01 to FF. It uses 10 locations from 00E0 to 00E9. The quotient appears in the lowest 5, with a fixed decimal implied between E1 and E2. The range of quotients is from $ 0000.010101 (from division of 0001/FF) to $ FFFF.000000 (from division of FFFF/01). Quotient locations are initially zeroed by a JSR to SUBROUTINE ZEROER, which must also be in memory and is coded separately for use in other programs. Before the JSR DIVIDA, 4 locations must be filled by the calling program. The dividend high byte is set in E6, the low byte in E7, and the divisor in E8. The "precision byte", with a value from 01 to 05, is set in location E9; it is not altered by the program, but the other 3 bytes usually are. The purpose of the precision byte is to allow the user to control the number of quotient locations to be calculated by DIVIDA. A value of 01 causes exit after the proper quotient value in location E0 (which may be 00) has been calculated. A value of 02 limits the calculation to quotient locations E0 and E1, and gives "integer arithmetic". A value of 03 allows only one location to the right of the implied decimal, etc.. The chief use is to shorten the execution time, which can approach 2000 microseconds at a precision of 05. However, DIVIDA always exits when the calculated remainder is zero, since calculation of higher-precision locations is then unnecessary. No "rounding-off" operations are included. E.g., the quotient of FEFE/FF is 00FF.FD0000 at a precision of 03, although it should be 00FF.FE since the quotient is 00FF.FDFDFD at a precision of 05.

DIVIDA exits in less than 150 microseconds if the dividend is 0000. It provides no protection against a divisor of 00, so the calling program should guard against this! A guard could be inserted in DIVIDA, but I feel it is better for the calling program to decide what should be done if such an error occurs.

Operation of DIVIDA involves addition of a shifting single-bit "Bit-Byte" in location 05, to the quotient location controlled by the X-register, whenever a positive remainder is obtained. The X-register is not protected by DIVIDA, so it is better to use Y-indexed loops in the calling program (that otherwise will have to store and restore the X value). The final remainder is in location E6 when DIVIDA exits. The divisor value is not altered if it is $ 80 or more; otherwise it is left shifted by DIVIDA.

DIVIDA is very long (70 bytes, or 78 if one includes ZEROER; if the zeroing operation were made an integral part of DIVIDA the length would be 74 bytes and execution a shade faster). It is also slow compared to hardware arithmetic, but relatively inexpensive. It is meant to handle data, that are never precise, and not the kind of complex math for which calculators are designed. Since the ROR instruction is not used, it will run in any 6502 system.

Much of the length of DIVIDA is caused by special logic designed to reduce the execution time---a deliberate trade-off of more program bytes for a lower average time, that has the effect of prolonging the time of divisions wher no early exit is possible.

Execution time depends both on the number of quotient locations to be filled and on the number of 1-bits to be inserted. Thus FFFF/01 runs slowly because it requires insertion of 16 1-bits into two locations. The "hi/lo exchange" operation at 0228 speeds up many operations with a dividend of 00XX. In general, higher speed will require sacrificing precision, and a precision-byte of 04 will be adequate. My reason for limiting the dividend to 16 bits and the divisor to 8 bits was that data more precise than 1 part in 256 will be rare, so that most data will be single-byte, and data sets with more than 256 items will be uncommon. Calculation of the average of 255 one-byte data items is within the capacity of DIVIDA. When there are more, they can be divided into subsets of 255 or fewer, the averages for all subsets added, and the average of the set of subsets calculated. We are now in the time range of seconds! With more bits, it would be minutes. People who need arithmetic speed had better get a 16-bit microprocessor (or better still, shell out for hardware multiply-divide).

Those who want integer arithmetic operations will do better using a dividend of type XX00 and precision-byte of 01. However, similar effects can usually be obtained more quickly and by other logic, not division. The number of possible ways of doing division is incredibly large, but I will be surprised if an operation like that of DIVIDA can be done with many fewer bytes or much higher speed, although using the ROR instruction might help.

SUBROUTINE ZEROER

```
0200 A9 00      (LDA# 00)
     95 DF      (STA zero-page, X)
0204 CA         (DEX)
     D0 FB      (BNE, if ≠ 0, back to 0202)
0207 60         (RTS)
```

SUBROUTINE DIVIDA

(Note that 3 locations are unused between the end of ZEROER and the start of DIVIDA. This is to allow users (if the subroutines are in RAM) to insert 3 instructions following the LDA divisor instruction at 0213. If the divisor is 00, DIVIDA is wrong. The instructions D0 01 00 substitute for this a BREAK to 1C00. If something more complex is needed, the 3 instructions can be a JMP or JSR to a longer sequence of instructions.)

```
020B A2 06      (LDA# 06)
     20 00 02   (JSR ZEROER, to zero 00E0 to 00E5)

0210 38         (SEC)
     26 E5      (ROL sets Bit-Byte to 01 and clears carry)
0213 A5 E8      (LDA divisor byte)
     30 05      (BMI, if bit 7 = 1, skip to 021C)
0217 26 E5      (ROL Bit-Byte)
     0A         (ASL, left-shift divisor in accumulator)
021A D0 F9      (BNE, if ≠ 0, back to BMI at 0215)
     85 E8      (STA bit-pattern 1XXX XXXX into divisor location)

021E A5 E6      (LDA dividend-hi)
     B0 0F      (BCS, if carry set, go to subtraction at 0231)
0222 D0 09      (BNE, if ≠ 0, go to CMP at 022D)
     A5 E7      (LDA dividend-lo)
0226 F0 28      (BEQ, dividend = 0 so exit to 0250)
     85 E6      (STA dividend-lo into dividend-hi location)
022A 86 E7      (STX zeros dividend-lo)
     E8         (INX to shift to next higher quotient location)

022D C5 E8      (CMP dividend-hi with divisor)
     90 0B      (BCC, divisor too large, bypass to 023C)
0231 E5 E8      (SBC, subtract divisor from dividend-hi)
     85 E6      (STA remainder into dividend-hi)
0235 18         (CLC for addition)
     B5 E0      (LDA zero-page, X the proper quotient byte)
0238 65 E5      (ADC the Bit-Byte)
     95 E0      (STA zero-page, X back into quotient location)

023C 46 E5      (LSR the Bit-Byte)
     D0 09      (BNE, if ≠ 0, bypass resetting)
0240 E8         (INX to shift to next higher quotient location)
     E4 E9      (CPX to precision-byte)
0243 F0 0B      (BEQ, if equal exit to 0250)
     A9 80      (LDA# 80 to reset)
0247 85 E5      (STA into E5 resets Bit-Byte)
```

```
0249 06 E7      (ASL dividend-lo starts dividend left-shift)
     26 E6      (ROL dividend-hi completes the shift)
024D 4C 1E 02   (JMP to 021E for next test sequence)

0250 60         (RTS)
```

16 BIT SQUARE ROOT

Here is a program which takes the square root of a 16 bit binary number and yields an eight bit integer plus eight bit binary fraction result. This routine was translated by J.B. Ross from an 8080 program written by R.E. DuPuy. The program is written as a subroutine and communicates with other programs via memory locations. All cpu registers are changed by this routine. Input and output data are located as follows:

```
8 bit input (high)       $00E0
8 bit input (low)        $00E1

8 bit output (integer)   $00E0
8 bit output (faction)   $00E1

other locations used are : $00E2 - $00E8
```


SQUARE ROOT SUBROUTINE

```
0100 A9 00    SQRT  LDA#  $00     initialize extended argument
0102 85 E2          STAZ  $E2
0104 85 E3          STAZ  $E3
0106 A9 FF          LDA#  $FF     initialize complemented result
0108 85 E4          STAZ  $E4
010A 85 E5          STAZ  $E5
010C A9 10          LDA#  $10     initialize loop count
010E 85 E8          STAZ  COUNT
0110 A2 02    LOOP  LDX#  $02     double left shift of  E2-E3-E0-E1
0112 06 E1    SHFT  ASLZ  $E1
0114 26 E0          ROLZ  $E0
0116 26 E3          ROLZ  $E3
0118 26 E2          ROLZ  $E2
011A CA             DEX
011B D0 F5          BNER  SHFT
011D 06 E5          ASLZ  $E5     shift partial result left
011F 26 E4          ROLZ  $E4
0121 E6 E5          INCZ  $E5     shift in a one on the right
0123 A5 E4          LDAZ  $E4     make a copy of shifted partial result
0125 85 E6          STAZ  $E6
0127 A5 E5          LDAZ  $E5
0129 85 E7          STAZ  $E7
012B 06 E7          ASLZ  $E7     shift copy of partial result left
012D 26 E6          ROLZ  $E6
012F E6 E7          INCZ  $E7     shift in a one on the right
0131 18             CLC           subtract shifted partial result from
0132 A5 E7          LDAZ  $E7       high 16 of current remainder (by
0134 65 E3          ADCZ  $E3       adding complement)
0136 85 E7          STAZ  $E7
0138 A5 E6          LDAZ  $E6
013A 65 E2          ADCZ  $E2
013C 85 E6          STAZ  $E6
013E 90 0A          BCCR  NOGO    test subtraction result
0140 C6 E5          DECZ  $E5     tack a zero onto complemented result
0142 A5 E7          LDAZ  $E7     replace high order 16 of current
0144 85 E3          STAZ  $E3       remainder with subtraction result
0146 A5 E6          LDAZ  $E6
0148 85 E2          STAZ  $E2
014A C6 E8    NOGO  DECZ  COUNT   decrement and test loop count
014C F0 03          BEQR  DONE
014E 18             CLC           jump to loop
014F 90 BF          BCCR  LOOP
0151 A5 E4    DONE  LDAZ  $E4     complement result and store in E0-E1
0153 49 FF          EOR#  $FF
0155 85 E0          STAZ  $E0
0157 A5 E5          LDAZ  $E5
0159 49 FF          EOR#  $FF
015B 85 E1          STAZ  $E1
015D 60             RTS           return
```

KIM Goes to the Moon

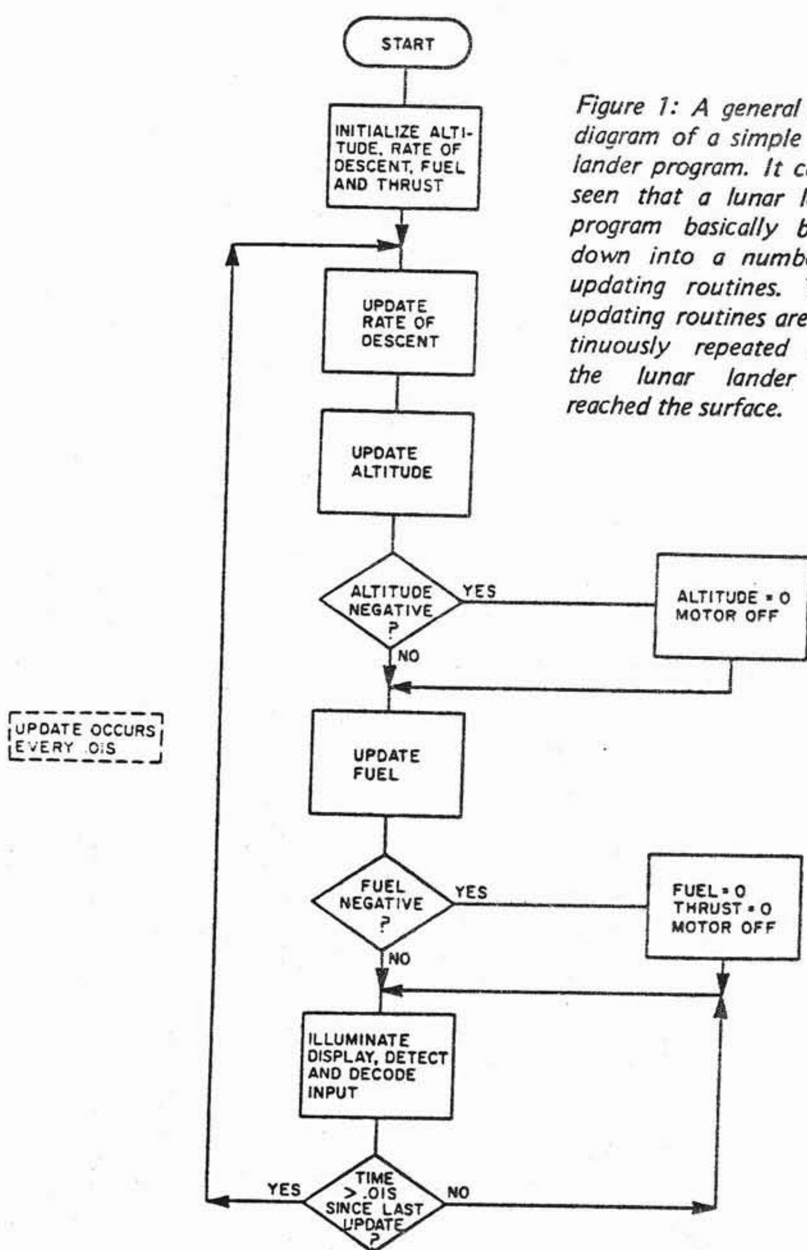


Figure 1: A general block diagram of a simple lunar lander program. It can be seen that a lunar lander program basically breaks down into a number of updating routines. These updating routines are continuously repeated until the lunar lander has reached the surface.

Jim Butterfield
14 Brooklyn Av
Toronto Ontario M4M 2X5 CANADA

There are quite a few lunar landing programs available nowadays: some for pocket calculators, others using graphic displays. The one I wrote for my KIM-1, based on the MOS Technology 6502 microprocessor, illustrates many of the techniques needed to develop the program.

The KIM-1 comes with a six digit LED display, which can be accessed by the user. I used the first four digits to represent the craft's altitude, and optionally, the fuel remaining. The last two digits, which are slightly separated from the rest of the display, are used for rate of descent. Both values change continually as the craft moves.

The KIM-1 keyboard is used as the pilot's control panel. Thrust is set by pressing controls 1 to 9. A value of 1 is minimum thrust, and the craft's rate of descent will increase due to gravity. Nine is maximum thrust, which slows the rate of descent sharply. In addition to power control, the pilot can elect to view either current altitude, by pressing A, or remaining fuel, by pressing F.

The Equations of Motion

The craft, of course, moves in accordance with the forces acting upon it: thrust and gravity. A physics textbook shows some rather formidable equations. However, they can be boiled down to the following simple procedure:

Every 0.01 second,
- add 0.01 of the acceleration to the velocity;
- add 0.01 of the velocity to the altitude;
- subtract 0.01 of the thrust from the fuel.

The acceleration is set equal to thrust minus gravity, and gravity is set at the constant value 5.

The time period of 0.01 s is arbitrary. Since KIM can operate in decimal mode, dividing by 100 becomes an elementary operation. Everything would work just as well if it were done in any other small time increment.

Figure 1 shows an elementary block diagram of the program. After setting the initial flight values, we settle into three main jobs: updating the flight, lighting the display, and detecting input from the pilot.

Setting Initial Values

An interesting flight can be obtained by starting the lunar module at a height of 4,500 feet with 800 pounds of fuel. That's more than sufficient fuel for a safe landing, but not enough to allow for prolonged hovering.

It's not difficult to set all the initial values by programming them individually. However, a faster method is to set them all together in memory and use a loop to initialize all of them. This is what I did as shown in listing 1 on hexadecimal lines 0000 to 0007.

Updating the Flight

Every 0.01 s we must update our rate of descent, altitude and fuel. As previously indicated, we have to add 0.01 of various values into the totals. We can accomplish this quite easily by using a gimmick. Instead of holding the altitude, for example, in feet, let's use two more digits and store it as multiples of 0.01 feet. Now we can add the rate of ascent directly into the six digit number; and the division by 100 happens automatically. For display purposes, of course, we drop the last two digits, so that we're back to height in feet. Using the same technique on the other parameters, we find that the updating job becomes relatively easy.

During the updating task, we must also detect two special conditions: touchdown and out of fuel. This seems fairly simple

Listing 1: An example lunar lander program written for the KIM-1 microprocessor that uses the flowchart of figure 1 as a base. The input and output of this program is handled by routines that are inherent to the KIM-1 system. The data display is seen on the keypad and LED display of the KIM-1 assembly. This display continuously shows the rate of descent, and on command will display either the amount of fuel left, or the altitude of the craft. Keys 1 through 9 are used to input thrust commands, while key A chooses the altitude display mode and the F key chooses the fuel display mode. All the numbers in this listing are in hexadecimal unless otherwise stated.

Address	Op	Operand	Label	Mnemonic	Commentary
0000	A2	0C	GO	LDX #0C	
0002	B5	B8	LP1	LDA INIT,X	initialize values;
0004	95	E2		STA ALT,X	
0006	CA			DEX	
0007	10	F9		BPL LP1	
0009	A2	05	CALC	LDX #05	X:=05;
000B	A0	01	RECAL	LDY #01	Y:=01;
000D	F8			SED	set decimal mode;
000E	18			CLC	clear carry;
000F	B5	E2	DIGIT	LDA ALT,X	
0011	75	E4		ADC ALT+2,X	add each digit;
0013	95	E2		STA ALT,X	
0015	CA			DEX	
0016	88			DEY	set up next digit;
0017	10	F6		BPL DIGIT	
0019	B5	E5		LDA ALT+3,X	
001B	10	02		BPL INCR	
001D	A9	99		LDA #99	
001F	75	E2	INCR	ADC ALT,X	
0021	95	E2		STA ALT,X	
0023	CA			DEX	counter:=counter - 1;
0024	10	E5		BPL RECAL	if counter positive go to RECAL;
0026	A5	E2		LDA ALT	else check if altitude is positive;
0028	10	08		BPL UP	if altitude positive go to UP;
002A	A9	00		LDA #00	else altitude:=00;
002C	A2	02		LDX #02	X:=02
002E	95	E2	DD	STA ALT,X	
0030	95	E8		STA TH2,X	else turn off engine;
0032	CA			DEX	
0033	10	F9		BPL DD	
0035	38		UP	SEC	set carry;
0036	A5	ED		LDA FUEL+2	
0038	E5	EA		SBC THRUST	update fuel;
003A	85	ED		STA FUEL+2	
003C	A2	01		LDX #01	
003E	B5	EB	LP2	LDA FUEL,X	
0040	E9	00		SBC #00	
0042	95	EB		STA FUEL,X	check if fuel left;
0044	CA			DEX	
0045	10	F7		BPL LP2	
0047	B0	0C		BCS TANK	if fuel left go to TANK;
0049	A9	00		LDA #00	
004B	A2	03		LDX #03	
004D	95	EA	LP3	STA THRUST,X	else turn off engine;
004F	CA			DEX	
0050	10	FB		BPL LP3	
0052	20	AA 00		JSR THRSET	go to THRSET;
0055	A5	EE	TANK	LDA MODE	A:=display mode;
0057	D0	0A		BNE SHOFL	if mode not 00 go to SHOFL;
0059	A5	E2		LDA ALT	AX:=location of altitude;
005B	A6	E3		LDX ALT+1	
005D	F0	08		BEQ ST	go to ST;
005F	D0	06		BNE ST	
0061	F0	A6	LINK	BEQ CALC	
0063	A5	EB	SHOFL	LDA FUEL	A:=FUEL;
0065	A6	EC		LDX FUEL+1	X:=FUEL+1;
0067	85	FB	ST	STA POINTH	display values;
0069	86	FA		STX POINTL	
006B	A5	E5		LDA VEL	A:=velocity sign;
006D	30	06		BMI DOWN	if sign negative go to DOWN;
006F	A5	E6		LDA VEL+1	A:=/velocity/;
0071	F0	07		BEQ FLY	go to FLY;
0073	D0	05		BNE FLY	
0075	38		DOWN	SEC	
0076	A9	00		LDA #00	velocity:=/velocity/;
0078	E5	E6		SBC VEL+1	

Listing 1, continued:

Address	Op	Operand	Label	Mnemonic	Commentary
007A	85	F9	FLY	STA INH	
007C	A9	02		LDA =02	} DECK:=02; [counter]
007E	85	E1		STA DECK	
0080	20	1F 1F	FLITE	JSR SCANDS	look for depressed key;
0083	F0	06		BEQ NOKEY	if no input go to NOKEY;
0085	20	6A 1F		JSR GETKEY	else go to GETKEY;
0088	20	91 00		JSR DOKEY	go to DOKEY;
008B	C6	E1	NOKEY	DEC DECK	DECK:=DECK-1;
008D	D0	F1		BNE FLITE	if DECK not equal to 0 go to FLITE;
008F	F0	D0		BEQ LINK	else go to LINK;
0091	C9	15	DOKEY	CMP =15	A:=fuel mode?;
0093	D0	03		BNE NALT	if not fuel mode go to NALT;
0095	85	EE		STA MODE	else MODE:= fuel mode;
0097	60			RTS	return;
0098	C9	10	NALT	CMP =10	A:=altitude mode?;
009A	D0	05		BNE NAL2	if not go to NAL2;
009C	A9	00		LDA =00	else mode:=altitude mode;
009E	85	EE		STA MODE	MODE:=A;
00A0	60		RET1	RTS	return;
00A1	10	FD	NAL2	BPL RET1	return; [illegal mode]
00A3	AA			TAX	else X:=A;
00A4	A5	EA		LDA THRUST	A:=THRUST;
00A6	F0	F8		BEQ RET1	if thrust:=0 go to RET1;
00A8	86	EA		STX THRUST	else THRUST:=X;
00AA	A5	EA	THRSET	LDA THRUST	A:=THRUST;
00AC	38			SEC	set carry;
00AD	E9	05		SBC =05	THRUST:=THRUST - 05;
00AF	85	E9		STA TH2+1	TH2+1:=THRUST;
00B1	A9	00		LDA =00	} A:=00;
00B3	E9	00		SBC =00	
00B5	85	E8		STA TH2	TH2:=00;
00B7	60			RTS	return;
00B8	45		INIT		
00B9	00				} [initial height]
00BA	00				
00BB	99				
00BC	80				} [initial speed]
00BD	00				
00BE	99				} [initial acceleration]
00BF	98				
00C0	02				[initial thrust]
00C1	08				
00C2	00				} [initial fuel]
00C3	00				
00C4	00				[mode]

until we realize that both the altitude and the fuel gauge will probably go right past the zero mark, jumping directly from a positive to a negative value; so a zero test is out. Instead, we take action the instant the number goes negative, restoring it to zero and then taking whatever other action is called for.

Lighting the Display

The display is quite straightforward; in fact, the KIM-1 monitor program has a subroutine to do the job.

Depending on the display mode flag, all we need to do is to move altitude or fuel to the display area, together with rate of descent. Then we call the subroutine to transfer it to the LEDs.

Of course, we must remember to drop the last two digits from the displayed values (0.01 of units, remember?) and to negate the rate of descent, where necessary, so that it shows as a positive number.

Detecting Input

The KIM-1 monitor subroutine that lights the display gives us a free bonus: It also tells us whether or not a key is depressed on the keyboard. To find out which key, we must call another subroutine in the monitor program.

If we discover that the user has input a thrust command, buttons 1 to 9, we first check to see that the motor is on and that we have fuel. Then we set the thrust, and also calculate the acceleration as thrust minus 5, where 5 represents the force of gravity.

The two other legal keys, A and F, set the display mode to altitude or fuel. The program sets a memory location which will be tested by the display routine.

The program doesn't need to worry about when a button is released. Although the question can be quite important for programs that must distinguish between, say, 9 and 99 on the input, the lunar lander doesn't really care. If you leave your finger on the button, it will keep on setting the thrust over and over to the same value, without affecting the flight.

Coming Down

The program doesn't stop. If you run out of fuel, you will watch yourself freefall to the surface. When you land, with or without fuel, your rate of descent freezes so that you can see how hard you landed.

It would be easy to have the display change after you land, to show words such as "SAFE" or "DEAD." The KIM-1 display is segment driven so that you can easily produce special combinations.

The novice astronaut who would like to try his or her hand at flying this, or other, craft should keep the following rules in mind:

1. Always conserve fuel at the beginning by reducing power to minimum thrust.
2. Don't let your rate of descent get excessively high; with my program, it's wise to steady up with a thrust value of 5 when your speed gets over 90 feet per second.
3. As you get to lower altitudes, try to balance your altitude against your rate of descent. At 1000 feet, a rate of descent of 500 feet per second will bring you down in 20 seconds, which is reasonable. Keep that sort of balance.■

HORSERACE

KIM-1 User Notes v1 #3
Charles K. Eaton
19606 Gary Ave.
Sunnyvale, CA 94086

Eight lap horse race and you can be the jockey and whip your horse to go faster. Warning--whip the horse too much and he probably poops out.

Horse	Track	Whipping button
Prince Charming	top	PC
Colorado Cowboy	middle	C
Irish Rair	bottom	4

Start program at 027F. Race is eight laps.

HORSE RACE

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0270	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	D8
0280	A2	13	BD	7C	03	95	7C	CA	10	F8	A9	7F	8D	41	17	A0
0290	00	A2	09	B9	7C	00	84	FC	20	4E	1F	C8	C0	06	90	F3
02A0	20	3D	1F	A5	8F	30	E3	A2	03	CA	30	DE	D6	86	D0	F9
02B0	86	99	A4	99	B6	83	B9	90	03	35	7C	EA	EA	EA	EA	EA
02C0	95	7C	E8	96	83	B9	90	03	49	FF	15	7C	95	7C	E0	05
02D0	30	38	D0	06	A5	8F	F0	28	D0	30	A2	02	38	B5	83	E9
02E0	06	95	83	CA	10	F6	A2	06	B5	7C	95	76	A9	80	95	7C
02F0	CA	D0	F5	EA	EA	EA	EA	EA	EA	EA	EA	EA	EA	EA	EA	EA
0300	C6	8F	D0	06	A5	81	09	06	85	81	EA	EA	EA	EA	EA	EA
0310	B9	89	00	F0	0B	20	68	03	29	3C	D0	1B	99	89	00	EA
0320	20	68	03	29	38	85	9A	B9	8C	00	30	0B	29	38	C5	9A
0330	B0	05	A9	FF	99	86	00	20	3D	1F	A0	FF	A6	99	3D	93
0340	03	F0	01	88	98	55	89	85	9A	EA	EA	20	68	03	38	29
0350	01	65	9A	18	A6	99	75	8C	EA	EA	EA	EA	EA	EA	EA	EA
0360	95	8C	95	86	4C	A9	02	38	38	A5	92	65	95	65	96	85
0370	91	A2	04	B5	91	95	92	CA	10	F9	60	80	80	80	80	80
0380	80	80	80	FF	FF	FF	80	80	80	00	00	00	80	80	80	08
0390	FE	BF	F7	01	02	04										

ONE-ARMED BANDIT

J. Butterfield
Toronto

Start program at $0200
Press any key to spin wheels

```
0200 A9 25      GO     LDA#   $25
0202 85 05             STAz   AMT
0204 20 BA 02          JSR@   CVAMT     CHANGE TO DISP
0207 A9 00             LDA#   $00
0209 85 06             STAz   ARROW
                :  MAIN DISPLAY LOOP
020B 20 8D 02   LP1    JSR@   DISPLY
020E D0 FB             BNEr   LP1
0210 E6 09      ROLL   INCz   TUMBLE
0212 20 8D 02          JSR@   DISPLAY
0215 F0 F9             BEQr   ROLL
0217 A9 03             LDA#   $03
0219 85 06             STAz   ARROW
021B F8                SED
021C 38                SEC
021D A5 05             LDAz   AMT
021F E9 01             SBC#   $01       CHARGE A BUCK
0221 85 05             STAz   AMT
0223 20 BA 02          JSR@   CVAMT
0226 26 09             ROLz   TUMBLE
0228 20 8D 02   LP2    JSR@   DISPLY
022B C6 08             DECz   STALL1    WAIT
022D D0 F9             BNEr   LP2
022F A6 06             LDXz   ARROW
0231 A5 09             LDAz   TUMBLE
0233 29 06             AND#   $36
0235 09 40             ORA#   $40       SPIN RESULT
0237 95 01             STAzx  WINDOW+1    TO DISPLAY
0239 46 09             LSRz   TUMBLE
023B 46 09             LSRz   TUMBLE
023D C6 06             DECz   ARROW
023F D0 E7             BNEr   LP2
                :  ALL WHEELS STOPPED - COMPUTE PAYOFF
0241 A5 04             LDAz   WINDOW+4
0243 C5 03             CMPz   WINDOW+3
0245 D0 37             BNEr   NOMAT
0247 C5 02             CMPz   WINDOW+2
0249 D0 33             BNEr   NOMAT
024B A2 10             LDX#   $10
024D C9 40             CMP#   $40       $15 IF 3 BARS
024F F0 0D             BEQr   PAY
0251 A2 0B             LDX#   $0B
0253 C9 42             CMP#   $42       $10 IF 3 UPS
0255 F0 07             BEQr   PAY
0257 A2 06             LDX#   $06
0259 C9 44             CMP#   $44       $5 IF 3 DOWNS
025B F0 01             BEQr   PAY
025D CA                DEX
```

```
                :  A WIN!  PAY AMOUNT IN X
025E 86 07      PAY    STXz   RWD
0260 A9 80      PAX    LDA#   $80
0262 85 08             STAz   STALL1
0264 20 8D 02   LP9    JSR@   DISPLY
0267 C6 08             DECz   STALL1
0269 D0 F9             BNEr   LP9
026B C6 07             DECz   RWD
026D F0 9C             BEQr   LP1
026F 18                CLC
0270 F8                SED
0271 A5 05             LDAz   AMT
0273 69 01             ADC#   $01
0275 B0 94             BCSr   LP1
0277 85 05             STAz   AMT
0279 20 BA 02          JSR@   CVAMT
027C D0 E2             BNEr   PAX
                :  WHEELS NOT ALL SAME - CHECK FOR SMALL WIN
027E A2 03      NOMAT  LDX#   $03
0280 C9 46             CMP#   $46       CHERRY
0282 F0 DA             BEQr   PAY
0284 20 8D 02   LOK    JSR@   DISPLY
0287 A5 05             LDAz   AMT
0289 D0 80             BNEr   LP1
028B F0 F7             BEQr   LOK
```

```
                :
                :  DISPLAY SUBROUTINE
                :
028D A6 06      DISPLY LDXz   ARROW
028F 10 02             BPLr   INDIS
0291 F6 02      OVER   INCzx  WINDOW+2
0293 CA         INDIS  DEX
0294 10 FB             BPLr   OVER
0296 A9 7F             LDA#   $7F
0298 8D 41 17          STA@   PADD
029B A0 0B             LDY#   $0B
029D A2 04             LDX#   $04
029F B5 00      LITE   LDAzx  WINDOW
02A1 8C 42 17          STY@   SBD
02A4 8D 40 17          STA@   SAD
02A7 D8                CLD
02A8 A9 7F             LDA#   $7F
02AA E9 01      ZIP    SBC#   $01
02AC D0 FC             BNEr   ZIP
02AE 8D 42 17          STA@   SBD
02B1 C8                INY
02B2 C8                INY
02B3 CA                DEX
02B4 10 E9             BPLr   LITE
02B6 20 40 1F          JSR@   KEYIN
02B9 60                RTS
                :
                :  AMOUNT CONVERSION
                :
02BA A5 05      CVAMT  LDAz   AMT
02BC 29 0F             AND#   $0F
02BE AA                TAX
02BF BD E7 1F          LDA@x  TABLE
02C2 85 00             STAz   WINDOW
02C4 A5 05             LDAz   AMT
02C6 4A                LSRa
02C7 4A                LSRa
02C8 4A                LSRa
02C9 4A                LSRa
02CA AA                TAX
02CB BD E7 1F          LDA@x  TABLE
02CE 85 01             STAz   WINDOW+1
02D0 60                RTS
```

Stan Ockers
Jim Butterfield

KIMMAZE

Find your way out of the maze. You're the flashing light in the center of the display. As you move up (key 9), down (key 1), left (key 4), or right (key 6), KIM will keep you in the central display; you'll see the walls of the maze moving by as you travel. Like walking through a real maze, you'll only see a small part of the maze as you pass through it. If you can get out, you'll find yourself in a large open area; that means you've won.
Program starts at address 0200.

```
0200 D8         START   CLD
0201 A2 02              LDX#    2         3 values
0203 BD B5 02   SETUP   LDA@x   INIT       from init
0206 95 D2              STAzx   MZPT        ..to maze ptr
0203 CA                 DEX
0209 10 F8              BPLr    SETUP
                ;--pick out specific part of maze
020B A0 0B      MAP     LDY#    11
020D B1 D2      GETMOR  LDAiy   MZPT      6 rows X 2
020F 99 D8 00           STAay   WORK
0212 88                 DEY
0213 10 F8              BPLr    GETMOR
                ;--shift to position vertically
0215 A2 0A              LDX#    10        for each of 6 rows..
0217 A4 D4      NXDIG   LDYz    POSIT      shift Y positions
0219 A9 FF              LDA#    $FF        filling with 'walls'
021B 38         REROL   SEC                ..on both sides
021C 36 D9              ROLzx   WORK+1
021E 36 D8              ROLzx   WORK       roll 'em
0220 2A                 ROLa
0221 88                 DEY
0222 D0 F7              BNEr    REROL
                ;--calculate segments
0224 29 07              AND#    7         take 3 bits
0226 A8                 TAY               & change to
0227 B9 A0 02           LDA@y   TAB1      segment pattern
022A 95 D8              STAzx   WORK      ..and store
022C CA                 DEX
022D CA                 DEX
022E 10 E7              BPLr    NXDIG
                ;--test flasher
0230 C6 D5      LIGHT   DECz    PLUG      time out?
0232 10 0A              BPLr    MUG       ..no
0234 A9 05              LDA#    5         ..yes, reset
0236 85 D5              STAz    PLUG
0233 A5 DE              LDAz    WORK+6     ..and..
023A 49 40              EOR#    $40         ..flip..
023C 85 DE              STAz    WORK+6       ...flasher..
```

```
                        ;--light display
023E A9 7F      MUG     LDA#    $7F        open the gate
0240 8D 41 17           STA@    SADD
0243 A0 09              LDY#    $09
0245 A2 0A              LDX#    10
0247 B5 D8      SHOW    LDAzx   WORK       tiptoe thru..
0249 8D 40 17           STA@    SAD         ..the segments
024C 8C 42 17           STY@    SBD
024F C6 D6      ST1     DECz    STALL      ...pausing
0251 D0 FC              BNEr    ST1
0253 C8                 INY
0254 C8                 INY
0255 CA                 DEX
0256 CA                 DEX
0257 10 EE              BPLr    SHOW
                        ;--test new key depression
0259 20 40 1F           JSR@    KEYIN      set dir reg
025C 20 6A 1F           JSR@    GETKEY      key?
025F C5 D7              CMPz    SOK          ..same as last?
0261 F0 CD              BEQr    LIGHT
0263 85 D7              STAz    SOK         no, record it
                        ;--test which key
0265 A2 04              LDX#    4          5 items in table
0267 DD A8 02   SCAN    CMP@x   TAB2
026A F0 05              BEQr    FOUND
026C CA                 DEX
026D 10 F8              BPLr    SCAN
026F 30 BC              BMIr    LIGHT
0271 CA         FOUND   DEX
0272 30 8C              BMIr    START      go key?
                        ;--test if wall
0274 BC AD 02           LDY@x   TAB3
0277 B9 D8 00           LDA@y   WORK
027A 3D B1 02           AND@x   TAB4
027D D0 B1              BNEr    LIGHT
                        ;--move
027F CA                 DEX
0280 10 04              BPLr    NOTUP
0282 C6 D4              DECz    POSIT      upward move
0284 D0 85      MLINK   BNEr    MAP        l-o-n-g branch!
0286 D0 04      NOTUP   BNEr    SIDEWY
0288 E6 D4              INCz    POSIT      downward move
028A D0 F8              BNEr    MLINK
028C CA         SIDEWY  DEX
028D D0 06              BNEr    LEFT
028F C6 D2              DECz    MZPT       right move
0291 C6 D2              DECz    MZPT
0293 D0 EF              BNEr    MLINK
0295 E6 D2      LEFT    INCz    MZPT       left move
0297 E6 D2              INCz    MZPT
0299 D0 E9              BNEr    MLINK

                        ;--tables (hex listed)
TAB1    02A0    00 08 40 48 01 09 41 49
TAB2    02A8    13 09 01 06 04
TAB3    02AD    06 06 04 08
TAB4    02B1    01 08 40 40
```

```
                ;--sample maze follows
                ;--first 3 bytes are initial cursor pointer
INIT    02B5    B4 02 08
MAZE    02B8    FF FF 04 08 F5 7E 15 00 41 FE 5F 04
                51 7D 5D 04 51 B6 54 14 F7 D5 04 54
                7F 5E 01 00 FD FF 00 00 00 00 00 00
                00 00 00 00 00 00
```

Maze construction: every two bytes, starting at MAZE, represents a complete cross section of the maze; a one bit in any position represents a wall.

In the example above, the first cross section is FF FF (all one bits) - this would be an impassable section of wall. The next cross section (04 03) has only two pieces of wall in it, at positions 6 and 13. The zeros at the end represent the 'open space'.

MUSIC MACHINE

F. J. Butterfield
Toronto

Description

This program plays one or several tunes via the 'Audio Out' interface of KIM-1. Use the same connection as that for recording on cassette tape. If your tape recorder has a 'monitor' feature, you can listen to the tune as well as record it. Alternatively, an amplifier can be used to play the tune through a speaker.

How to Run

Load the program. Load the tune(s) from cassette or from the keyboard. Tunes start at location$0000. Be sure to store the value $FA at the end of each tune, and behind the last tune, store: $FF, $00. Since this program uses the Break instruction to transfer control back to the monitor after each tune is played, you must set up the software interrupt vector by storing $00 in $17FE, and $1C in $17FF.

The starting address for the program is $0200. To play the next tune, press GO.

How to Write your own Tune

Each note goes into a byte of storage, starting at location $0000 of memory. Each tune should end with the value $FA which stops the program until GO is pressed.

Special codes are incorporated in the program to allow certain effects - adjustment of speed, tone, etc. The codes are followed by a value which sets the particular effect. The codes are listed below:

Code	Effect	Initially	Examples
FB	sets speed of tune	$30	18 is quick; 60 is slow
FC	sets length of 'long'notes	$02	2 means 'long'note lasts twice as long'as'short'
FD	sets pitch	$01	2 is bass; 4 is deep bass
FE	sets instrument	$FF	FF is piano, 00 is clarinet
FF	sets address for tune	$00	00 will take you back to first tune; like a 'jump'

For example, at any time during a tune, you may insert the sequence $FB $18 and the tune will begin to play at a fast speed. Inserting $FF $45 will cause a switch to the tune at zero page address $0045. The initial values shown can be reset at any time by starting at address $0200.

No tune should extend beyond address $00DF, since program values are stored at $00E0 and up.

The program can be easily converted to a subroutine by replacing the BRK instruction with RTS. This allows the programmer to play various 'phrases' of music to produce quite complex tunes.

The lowest note you can play is A below middle C. You can play short notes and long notes (a long note is twice as long as a short note). If you want to stretch out a note even longer than a long note allows, put a 'pause' note after it. Some of the notes are as follows:

	Note	Short	Long
	A	79	F9
	A#	72	F2
	B	6C	EC
middle	C	66	E6
	C#	60	E0
	D	5A	DA
	D#	56	D6
	E	51	D1
	F	4C	CC
	F#	48	C8
	G	44	C4
	G#	40	C0
	A	3D	BD
	A#	39	B9
	B	36	B6
	C	33	B3
	C#	30	B0
	D	2D	AD
	E	28	A8
	F	26	A6
	PAUSE	00	80

Sample Tunes

```
0000 FB 18 FE FF 44 51 E6 E6 66 5A 51 4C C4 C4 C4 D1
0010 BD BD BD 00 44 BD 00 44 3D 36 33 2D A8 80 80 33
0020 44 B3 80 80 44 51 C4 80 80 5A 51 E6 80 80 FA FE
0030 00 FB 28 5A 5A 51 48 5A 48 D1 5A 5A 51 48 DA E0
0040 5A 5A 51 48 44 48 51 5A 60 79 6C 60 DA DA FA FE
0050 FF 5A 5A 5A 5A 5A 5A 66 72 79 E6 E6 80 00 56 56
0060 56 56 56 56 5A 66 F2 80 80 4C 4B 4C 4C 4C 4C 56
0070 5A 56 4C 00 C4 44 4C 56 5A 5A 56 5A 66 56 5A 66
0080 F2 80 FE 00 00 72 5A CC 72 5A CC 72 5A CC 80 B8
0090 80 4C 56 5A 56 5A E6 F2 80 FA FF 00
```

MUSIC MACHINE

F. J. Butterfield
Toronto

```
                  ; initialize
                  ;
0200 A2 05        START  LDX#   $05
0202 BD 86 02     LP1    LDA@X  INIT
0205 95 E0               STAZX  WORK
0207 CA                  DEX
0208 10 F8               BPLR   LP1
                  ;
                  ; main routine here - WORK not reset
                  ;
020A A9 BF        GO     LDA#   $BF
020C 8D 43 17            STA@   PBDD   open output channel
020F A0 00               LDY#   $00
0211 B1 E4               LDAIY  WORK+4    get next note
0213 E6 E4               INCZ   WORK+4
0215 C9 FA               CMP#   $FA    test for halt
0217 D0 04               BNER   NEXT
0219 00                  BRK           (or RTS if used as subroutine)
021A EA                  NOP
021B F0 ED               BEQR   GO     resume when GO pressed
021D 90 0B        NEXT   BCCR   NOTE   is it a note?
021F E9 FB               SBC#   $FB    if not, decode instruction
0221 AA                  TAX           and put into X
0222 B1 E4               LDAIY  WORK+4    get parameter
0224 E6 E4               INCZ   WORK+4    and
0226 95 E0               STAZX  WORK      store in work table
0228 B0 E0               BCSR   GO        jump to GO
                  ;
                  ; set up timing for note
                  ;
022A A6 E0        NOTE   LDXZ   WORK   timing
022C 86 E7               STXZ   LIMIT+1
022E A6 E1               LDXZ   WORK+1    long note factor
0230 A8                  TAY           test accumulator
0231 30 02               BMIR   OVER   long note?
0233 A2 01               LDX#   $01    nope, get short note
0235 86 E6        OVER   STXZ   LIMIT     store length factor
0237 29 7F               AND#   $7F    remove short/long flag
0239 85 E9               STAZ   VAL2
023B F0 02               BEQR   HUSH   is it a pause?
023D 85 EA               STAZ   VAL1   no, set pitch
023F A5 E9        HUSH   LDAZ   VAL2   get timing and
0241 25 E3               ANDZ   WORK+3    bypass if muted
0243 F0 04               BEQR   ON
0245 E6 EA               INCZ   VAL1   else fade the
0247 C6 E9               DECZ   VAL2   note
0249 A6 E9        ON     LDX#   VAL2
024B A9 A7               LDA#   $A7
024D 20 5D 02            JSR@   SOUND
0250 30 B8               BMIR   GO
0252 A6 EA               LDX#   VAL1
0254 A9 27               LDA#   $27
0256 20 5D 02            JSR@   SOUND
0259 30 AF               BMIR   GO
025B 10 E2               BPLR   HUSH
                  ;
                  ; subroutine to send a bit
                  ;
025D A4 E2        SOUND  LDYZ   WORK+2    octave flag
025F 84 EB               STYZ   TIMER
0261 86 EC               STXZ   XSAV
0263 E0 00        SLOOP  CPX#   $00
0265 D0 08               BNER   CONT
0267 A6 EC               LDXZ   XSAV
0269 C6 EB               DECZ   TIMER
026B D0 F6               BNER   SLOOP
026D F0 16               BEQR   SEX
026F 8D 42 17     CONT   STA@   SBD
0272 CA                  DEX
0273 C6 E8               DECZ   LIMIT+2
0275 D0 EC               BNER   SLOOP
0277 C6 E7               DECZ   LIMIT+1
0279 D0 E8               BNER   SLOOP
027B A4 E0               LDYZ   WORK
027D 84 E7               STYZ   LIMIT+1
027F C6 E6               DECZ   LIMIT
0281 D0 E0               BNER   SLOOP
0283 A9 FF               LDA#   $FF
0285 60           SEX    RTS
                  ;
                  ; initial constants
                  ;
0286 30
0287 02
0288 01
0289 FF
028A 00
028B 00
                  ;
                  ; work areas reserved
                  ;
00E0              WORK   *=* +6      speed/length ratio/octave/tone
00E6              LIMIT  *=* +3      timing of note
00E9              VAL2   *=* +1      marking and spacing
00EA              VAL1   *=* +1      durations
00EB              TIMER  *=* +1      octave counter
00EC              XSAV   *=* +1
```

HUNT THE WUMPUS

Game by Gregory Yob
Adapted for the KIM-1 by Stan Ockers

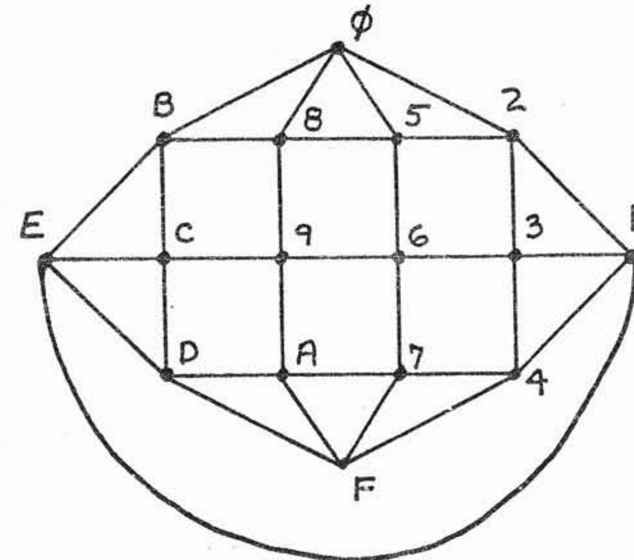
Stan Ockers
R.R. #4 Box 209
Lockport, Ill 60441

I first ran across the WUMPUS in THE BEST OF CREATIVE COMPUTING where it is programmed in basic. The following is based on this program with modifications so I could fit the program and messages in the KIM-1 memory. The messages appear on the display in scanning form with "sort-of" alphanumeric letters.

The WUMPUS lives in a cave of 16 rooms (labeled Ø - F). Each room has four tunnels leading to other rooms (see the map below). When the program is started, you and the WUMPUS are placed at random. Also placed at random are two bottomless pits (they don't bother the WUMPUS, he has sucker-type feet) and two rooms with SUPERBATS, (also no trouble to WUMPUS, he's too heavy). If you enter a room with a pit, you fall in and lose. If you enter a BAT'S room you are picked up and flown at random to another room. You will be warned when BATS, PITS, or the WUMPUS are nearby. If you enter the room with WUMPUS, he wakes and either moves to an adjacent room or just eats you up (you lose). In order to capture the WUMPUS and win, you must use "MOOD CHANGE" gas. When thrown into a room containing the WUMPUS, the gas causes him to turn from a vicious snarling beast into a meek and loveable creature. He will even come out and give you a hug. Beware though, you have only three cans of gas and once you toss a can of gas into a room it is contaminated and you cannot enter or you will be turned into beast (you lose)!

The program starts at $0300. If you lose and want everything to remain the same, (except the room you are in), resart at $0316. Use the reset key to stop the program because about half of page one is used and if you just use the ST key the stack will eventually work its way down into the program. The byte at $0229 controls the speed of the display. Once you get used to the characters you can speed things up by putting in a lower number. The message normally given tells you what room you are in and what the choices are for the next room. In order to fire the mood gas, press PC (pitch can) when the rooms to be selected are displayed. Then indicate the room into which you want to pitch the can. It takes a fresh can of gas to get the WUMPUS (he may move into a room already gassed). GOOD HUNTING!

CAVE MAP



APPENDIX D.

KIM DEMONSTRATION TAPE

KIM-1 DEMONSTRATION TAPE

Index	ID#	Name	Entry Point	Address Range
	01	DIRECTORY	\$1780	\$1780-\$17AF
	02	VU TAPE	\$0000	\$0000-\$0049
	03	SUPER TAPE (3X speed)	\$0100	\$0100-\$01C2
	04	MOVE A BLOCK	\$1780	\$1780-\$17CB
	05	HEDEC	\$0200	\$0200-\$0244
	06	ADC DEMONSTRATION - BINARY - BCD	\$0000 \$0020	\$0000-\$00A4
	07	FREQUENCY COUNTER	\$0000	\$0000-\$0067
	08	TAPE DUPE	\$1780	\$1780-\$17A9
	09	REAL TIME CLOCK	\$0370	\$0370-\$0400
	0A	STOP WATCH	\$0300	\$0300-\$0386
	10	LUNAR LANDER	\$0000	\$0000-\$00C6
	11	HORSE RACE	\$027F	\$027F-\$0396
	12	ONE ARMED BANDIT	\$0200	\$0200-\$02D1
	13	KIMAZE	\$0200	\$0200-\$02F0
	14	MUSIC MACHINE	\$0200	\$0000-\$028C
	15	HUNT THE WUMPUS	\$0300	\$0000-\$0400

Notes:

Supertape is set for 3X speed. To obtain the 6X speed change location \$01BE to \$02 and \$01C0 to \$03.

Move A Block uses data stored in memory as follows:

\$00E0	SAL old	\$00E1	SAH old
\$00E2	EAL old	\$00E3	EAH old
\$00E4	SAL new	\$00E5	SAL new

Frequency Counter: Connect IRQ to PB7. Signal input is PB0.

Music Machine: Be sure to set up the BRK vector by storing \$00 in \$17FE and \$1C in \$17FF.

Real Time Clock: Connect NMI to PB7, store \$A5 in \$17FA and \$03 in \$17FB. Restart display program at \$0379. Press "1" to return to the KIM monitor.

DIRECTORY

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1780	D8	A9	07	8D	42	17	20	41	1A	46	F9	05	F9	85	F9	C9
1790	16	D0	F3	20	24	1A	C6	F9	10	F5	C9	2A	D0	F1	A2	FD
17A0	20	F3	19	95	FC	E8	30	F8	20	1F	1F	D0	D3	F0	F9	

VU TAPE

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000	D8	A9	7F	8D	41	17	A9	13	85	E0	8D	42	17	20	41	1A
0010	46	F9	05	F9	85	F9	8D	40	17	C9	16	D0	E9	20	24	1A
0020	C9	2A	D0	F5	A9	00	8D	E9	17	20	24	1A	20	00	1A	D0
0030	D5	A6	E0	E8	E8	E0	15	D0	02	A2	09	86	E0	8E	42	17
0040	AA	BD	E7	1F	8D	40	17	D0	DB							

SUPER TAPE (3X)

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0100	A9	AD	8D	EC	17	20	32	19	A9	27	85	E1	A9	BF	8D	43
0110	17	A2	64	A9	16	20	61	01	A9	2A	20	88	01	AD	F9	17
0120	20	70	01	AD	F5	17	20	6D	01	AD	F6	17	20	6D	01	20
0130	EC	17	20	6D	01	20	EA	19	AD	ED	17	CD	F7	17	AD	EE
0140	17	ED	F8	17	90	E9	A9	2F	20	88	01	AD	E7	17	20	70
0150	01	AD	E8	17	20	70	01	A2	02	A9	04	20	61	01	4C	5C
0160	18	86	E0	48	20	88	01	68	C6	E0	D0	F7	60	20	4C	19
0170	48	4A	4A	4A	4A	20	7D	01	68	20	7D	01	60	29	0F	C9
0180	0A	18	30	02	69	07	69	30	A0	08	84	E2	A0	02	84	E3
0190	BE	BE	01	48	2C	47	17	10	FB	B9	BF	01	8D	44	17	A5
01A0	E1	49	80	8D	42	17	85	E1	CA	D0	E9	68	C6	E3	F0	05
01B0	30	07	4A	90	DB	A0	00	F0	D7	C6	E2	D0	CF	60	04	C3
01C0	06	7E														

MOVE A BLOCK

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1780	38	A5	E4	E5	E0	85	E6	A5	E5	E5	E1	85	E7	90	18	38
1790	A5	E2	E5	E0	A8	84	E8	E6	E8	B1	E0	91	E4	88	D0	F9
17A0	B1	E0	91	E4	88	30	14	38	A5	E2	E5	E0	85	E8	E6	E8
17B0	A0	00	B1	E0	91	E4	C8	C4	E8	D0	F7	18	A5	E2	65	E6
17C0	85	E2	A5	E3	65	E7	85	E3	4C	4F	1C					

HEDEC

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0200	F8	98	48	8A	48	A9	00	A2	06	95	DF	CA	D0	FB	E6	E5
0210	A5	E7	48	A0	08	68	4A	48	90	0C	A2	03	18	B5	E2	75
0220	DF	95	DF	CA	D0	F7	A2	03	18	B5	E2	75	E2	95	E2	CA
0230	D0	F7	88	D0	E0	68	A5	E3	D0	04	A5	E6	D0	D4	68	AA
0240	68	A8	D8	60												

ADC DEMONSTRATION

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000	A9	FF	8D	01	17	AD	03	17	29	EF	8D	03	17	20	80	00
0010	85	F9	20	1F	1F	4C	0D	00	00	00	00	00	00	00	00	00
0020	A9	FF	8D	01	17	AD	03	17	29	EF	8D	03	17	20	80	00
0030	85	E7	A2	00	86	E6	20	00	02	A6	E1	86	FB	A6	E2	86
0040	FA	A2	00	86	F9	20	1F	1F	4C	2D	00	00	00	00	00	00
0050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0080	A9	80	85	EE	A9	00	18	65	EE	8D	00	17	AD	02	17	29
0090	10	D0	09	AD	00	17	38	E5	EE	4C	9F	00	AD	00	17	46
00A0	EE	90	E3	60												

FREQUENCY COUNTER

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000	A9	01	85	65	F8	A9	36	8D	FE	17	A9	00	8D	FF	17	58
0010	00	EA	AD	02	17	29	01	D0	F9	AD	02	17	29	01	F0	F9
0020	18	A9	01	65	F9	85	F9	A9	00	65	FA	85	FA	A9	00	65
0030	FB	85	FB	4C	12	00	48	A9	90	8D	04	17	2C	07	17	10
0040	FB	A9	F4	8D	0F	17	C6	65	F0	02	68	40	A9	FF	85	66
0050	20	1F	1F	C6	66	D0	F9	A9	00	85	F9	85	FA	85	FB	A9
0060	05	85	65	68	40	03	00									

TAPE DUPE

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1780	A9	27	A2	3F	8E	43	17	A2	07	8E	42	17	A0	5E	2C	42
1790	17	10	02	A0	A3	A2	BF	8E	43	17	49	80	8D	42	17	8C
17A0	44	17	2C	47	17	10	FB	30	D9							

REAL-TIME CLOCK

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0370	A9	00	85	80	A9	F4	8D	0F	17	A5	81	85	F9	A5	82	85
0380	FA	A5	83	85	FB	20	6A	1F	C9	01	D0	0D	20	1F	1F	20
0390	6A	1F	C9	01	D0	03	4C	4F	1C	20	1F	1F	18	90	DA	00
03A0	00	00	00	00	00	48	8A	48	98	48	A9	83	8D	04	17	2C
03B0	07	17	10	FB	E6	80	A9	04	C5	80	D0	38	A9	00	85	80
03C0	18	F8	A5	81	69	01	85	81	C9	60	D0	28	A9	00	85	81
03D0	A5	82	18	69	01	85	82	C9	60	D0	19	A9	00	85	82	A5
03E0	83	18	69	01	85	83	C9	12	D0	02	E6	84	C9	13	D0	04
03F0	A9	01	85	83	D8	A9	F4	8D	0F	17	68	A8	68	AA	68	40

STOP WATCH

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0300	A9	76	8D	F2	17	A9	02	8D	F3	17	A9	00	85	F9	85	FA
0310	85	FB	20	1F	1F	20	6A	1F	C9	04	D0	03	4C	64	1C	C9
0320	03	D0	1F	A5	FB	20	3B	1E	A9	3A	20	A0	1E	A5	FA	20
0330	3B	1E	A9	2E	20	A0	1E	A5	F9	20	3B	1E	20	2F	1E	38
0340	B0	D0	C9	02	F0	C4	C9	01	D0	C8	A9	9C	8D	06	17	20
0350	1F	1F	AD	07	17	F0	FB	8D	00	1C	A9	9C	8D	06	17	18
0360	F8	A5	F9	69	01	85	F9	A5	FA	69	00	85	FA	C9	60	D0
0370	0B	A9	00	85	FA	A5	FB	18	69	01	85	FB	D8	20	6A	1F
0380	C9	00	D0	CB	F0	8C										

LUNAR LANDER

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000	A2	0C	B5	B8	95	E2	CA	10	F9	A2	05	A0	01	F8	18	B5
0010	E2	75	E4	95	E2	CA	88	10	F6	B5	E5	10	02	A9	99	75
0020	E2	95	E2	CA	10	E5	A5	E2	10	0B	A9	00	A2	02	95	E2
0030	95	E8	CA	10	F9	38	A5	ED	E5	EA	85	ED	A2	01	B5	EB
0040	E9	00	95	EB	CA	10	F7	B0	0C	A9	00	A2	03	95	EA	CA
0050	10	FB	20	AA	00	A5	EE	D0	0A	A5	E2	A6	E3	F0	08	D0
0060	06	F0	A6	A5	EB	A6	EC	85	FB	86	FA	A5	E5	30	06	A5
0070	E6	F0	07	D0	05	38	A9	00	E5	E6	85	F9	A9	02	85	E1
0080	20	1F	1F	F0	06	20	6A	1F	20	91	00	C6	E1	D0	F1	F0
0090	D0	C9	15	D0	03	85	EE	60	C9	10	D0	05	A9	00	85	EE
00A0	60	10	FD	AA	A5	EA	F0	F8	86	EA	A5	EA	38	E9	05	85
00B0	E9	A9	00	E9	00	85	E8	60	45	00	00	99	80	00	99	98
00C0	02	08	00	00	00	00										

HORSE RACE

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0270	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	D8
0280	A2	13	BD	7C	03	95	7C	CA	10	F8	A9	7F	8D	41	17	A0
0290	00	A2	09	B9	7C	00	84	FC	20	4E	1F	C8	C0	06	90	F3
02A0	20	3D	1F	A5	8F	30	E3	A2	03	CA	30	DE	D6	86	D0	F9
02B0	86	99	A4	99	B6	83	B9	90	03	35	7C	EA	EA	EA	EA	EA
02C0	95	7C	E8	96	83	B9	90	03	49	FF	15	7C	95	7C	E0	05
02D0	30	38	D0	06	A5	8F	F0	28	D0	30	A2	02	38	B5	83	E9
02E0	06	95	83	CA	10	F6	A2	06	B5	7C	95	76	A9	80	95	7C
02F0	CA	D0	F5	EA	EA	EA	EA	EA	EA	EA	EA	EA	EA	EA	EA	EA
0300	C6	8F	D0	06	A5	81	09	06	85	81	EA	EA	EA	EA	EA	EA
0310	B9	89	00	F0	0B	20	68	03	29	3C	D0	1B	99	89	00	EA
0320	20	68	03	29	38	85	9A	B9	8C	00	30	0B	29	38	C5	9A
0330	B0	05	A9	FF	99	86	00	20	3D	1F	A0	FF	A6	99	3D	93
0340	03	F0	01	88	98	55	89	85	9A	EA	EA	20	68	03	38	29
0350	01	65	9A	18	A6	99	75	8C	EA	EA	EA	EA	EA	EA	EA	EA
0360	95	8C	95	86	4C	A9	02	38	38	A5	92	65	95	65	96	85
0370	91	A2	04	B5	91	95	92	CA	10	F9	60	80	80	80	80	80
0380	80	80	80	FF	FF	FF	80	80	80	00	00	00	80	80	80	08
0390	FE	BF	F7	01	02	04										

ONE ARMED BANDIT

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0200	A9	25	85	05	20	BA	02	A9	00	85	06	20	8D	02	D0	FB
0210	E6	09	20	8D	02	F0	F9	A9	03	85	06	F8	38	A5	05	E9
0220	01	85	05	20	BA	02	26	09	20	8D	02	C6	08	D0	F9	A6
0230	06	A5	09	29	06	09	40	95	01	46	09	46	09	C6	06	D0
0240	E7	A5	04	C5	03	D0	37	C5	02	D0	33	A2	10	C9	40	F0
0250	0D	A2	0B	C9	42	F0	07	A2	06	C9	44	F0	01	CA	86	07
0260	A9	80	85	08	20	8D	02	C6	08	D0	F9	C6	07	F0	9C	18
0270	F8	A5	05	69	01	B0	94	85	05	20	BA	02	D0	E2	A2	03
0280	C9	46	F0	DA	20	8D	02	A5	05	D0	80	F0	F7	A6	06	10
0290	02	F6	02	CA	10	FB	A9	7F	8D	41	17	A0	0B	A2	04	B5
02A0	00	8C	42	17	8D	40	17	D8	A9	7F	E9	01	D0	FC	8D	42
02B0	17	C8	C8	CA	10	E9	20	40	1F	60	A5	05	29	0F	AA	BD
02C0	E7	1F	85	00	A5	05	4A	4A	4A	4A	AA	BD	E7	1F	85	01
02D0	60															

KIMAZE

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0200	D8	A2	02	BD	B5	02	95	D2	CA	10	F8	A0	0B	B1	D2	99
0210	D8	00	88	10	F8	A2	0A	A4	D4	A9	FF	38	36	D9	36	D8
0220	2A	88	D0	F7	29	07	A8	B9	A0	02	95	D8	CA	CA	10	E7
0230	C6	D5	10	0A	A9	05	85	D5	A5	DE	49	40	85	DE	A9	7F
0240	8D	41	17	A0	09	A2	0A	B5	D8	8D	40	17	8C	42	17	C6
0250	D6	D0	FC	C8	C8	CA	CA	10	EE	20	40	1F	20	6A	1F	C5
0260	D7	F0	CD	85	D7	A2	04	DD	A8	02	F0	05	CA	10	F8	30
0270	BC	CA	30	8C	BC	AD	02	B9	D8	00	3D	B1	02	D0	B1	CA
0280	10	04	C6	D4	D0	85	D0	04	E6	D4	D0	F8	CA	D0	06	C6
0290	D2	C6	D2	D0	EF	E6	D2	E6	D2	D0	E9	00	00	00	00	00
02A0	00	08	40	48	01	09	41	49	13	09	01	06	04	06	06	04
02B0	08	01	08	40	40	B4	02	08	FF	FF	04	08	F5	7E	15	00
02C0	41	FE	5F	04	51	7D	5D	04	51	B6	54	14	F7	D5	04	54
02D0	7F	5E	01	00	FD	FF	00	00	00	00	00	00	00	00	00	00
02E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

MUSIC MACHINE

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000	FB	18	FE	FF	44	51	E6	E6	66	5A	51	4C	C4	C4	C4	D1
0010	BD	BD	BD	00	44	BD	00	44	3D	36	33	2D	A8	80	80	33
0020	44	B3	80	80	44	51	C4	80	80	5A	51	E6	80	80	FA	FE
0030	00	FB	28	5A	5A	51	48	5A	48	D1	5A	5A	51	48	DA	E0
0040	5A	5A	51	48	44	48	51	5A	60	79	6C	60	DA	DA	FA	FE
0050	FF	5A	5A	5A	5A	5A	5A	66	72	79	E6	E6	80	56	56	56
0060	56	56	56	56	5A	66	D9	80	80	4C	4B	4C	4C	4C	4C	56
0070	5A	56	4C	00	C4	44	4C	56	5A	5A	56	5A	66	56	5A	66
0080	F2	80	FE	00	00	72	5A	CC	72	5A	CC	72	5A	CC	80	B8
0090	80	4C	56	5A	56	5A	E6	F2	80	FA	FF	00				

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0200	A2	05	BD	86	02	95	E0	CA	10	F8	A9	BF	8D	43	17	A0
0210	00	B1	E4	E6	E4	C9	FA	D0	04	00	EA	F0	ED	90	0B	E9
0220	FB	AA	B1	E4	E6	E4	95	E0	B0	E0	A6	E0	86	E7	A6	E1
0230	A8	30	02	A2	01	86	E6	29	7F	85	E9	F0	02	85	EA	A5
0240	E9	25	E3	F0	04	E6	EA	C6	E9	A6	E9	A9	A7	20	5D	02
0250	30	B8	A6	EA	A9	27	20	5D	02	30	AF	10	E2	A4	E2	84
0260	EB	86	EC	E0	00	D0	08	A6	EC	C6	EB	D0	F6	F0	16	8D
0270	42	17	CA	C6	E8	D0	EC	C6	E7	D0	E8	A4	E0	84	E7	C6
0280	E6	D0	E0	A9	FF	60	30	02	01	FF	00	00				

HUNT THE WUMPUS

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000	80	EE	DC	BE	80	F7	D0	F9	80	84	D4	80	DE	85	C0	80
0010	F8	BE	D4	D4	F9	B8	ED	80	B8	F9	F7	DE	80	F8	DC	85
0020	F7	B9	F9	F1	80	00	80	DC	DC	F3	ED	80	C0	80	FC	BE
0030	B7	F3	F9	DE	80	F7	80	9C	BE	B7	F3	BE	ED	80	80	00
0040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0050	02	02	00	01	01	00	03	04	00	06	07	00	09	0A	01	04
0060	05	03	01	02	03	02	05	06	05	08	09	08	0B	0C	0B	07
0070	08	04	03	04	07	06	07	0A	09	0A	0F	0C	0D	0E	0C	0A
0080	0B	0E	05	06	0F	08	09	0F	0B	0C	0D	0E	0E	0F	0D	0D
0090	80	B7	84	ED	ED	F9	DE	80	C0	80	DC	D4	B8	EE	80	DB
00A0	80	B9	F7	D4	ED	80	B8	F9	F1	F8	80	00	80	EE	DC	BE
00B0	80	B8	DC	ED	F9	80	00	80	D0	DC	DC	B7	D3	80	00	00

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0100	80	9C	BE	B7	F3	BE	ED	85	B9	B8	DC	ED	F9	00	80	F3
0110	84	F8	80	B9	B8	DC	ED	F9	00	80	FC	F7	F8	ED	80	B9
0120	D0	1E	D0	1E	D0	1E	48	22	0F	E6	63	1E	6C	1C	0F	E6
0130	63	1E	6C	1C	BD	DC	F8	80	EE	DC	BE	80	00	80	ED	BE
0140	F3	F9	D0	FC	F7	F8	80	ED	D4	F7	F8	B9	F6	80	00	80
0150	EE	EE	84	84	F9	F9	F9	80	F1	F9	B8	B8	80	84	D4	80
0160	F3	84	F8	80	00	80	BD	F7	ED	80	84	D4	80	D0	DC	DC
0170	B7	80	00	80	DC	BE	F8	80	DC	F1	80	BD	F7	ED	80	00

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0200	84	DE	85	DD	A9	07	85	DF	A0	05	A2	05	B1	DD	C9	00
0210	D0	01	60	95	E8	88	CA	10	F3	D8	18	98	65	DF	85	DC
0220	20	28	02	A4	DC	4C	0A	02	A2	06	86	DB	A9	52	8D	07
0230	17	20	3E	02	2C	07	17	10	F8	C6	DB	D0	EF	60	A9	7F
0240	8D	41	17	A0	00	A2	09	B9	E8	00	84	FC	20	4E	1F	C8
0250	C0	06	90	F3	20	3D	1F	60	20	8C	1E	20	3E	02	D0	F8
0260	20	3E	02	F0	FB	20	3E	02	F0	F6	20	6A	1F	C9	15	10
0270	E7	60	A5	C0	D0	04	E6	C0	D0	F8	29	8E	F0	05	0A	90
0280	FD	F0	05	06	C0	A5	C0	60	06	C0	E6	C0	A5	C0	60	A2
0290	04	D5	CB	F0	03	CA	10	F9	60	20	72	02	29	0F	C9	04
02A0	30	0D	20	B2	02	AD	06	17	29	03	AA	B5	C6	85	CB	A5
02B0	CB	60	A6	CA	B5	50	85	C6	B5	60	85	C7	B5	70	85	C8
02C0	B5	80	85	C9	60	A2	03	D5	C6	F0	03	CA	10	F9	60	A0
02D0	01	20	00	02	A0	00	A9	AC	20	00	02	4C	D4	02	BD	D0
02E0	F9	F7	F8	C0	80	EE	DC	BE	80	BD	F9	F8	80	F7	80	F6
02F0	BE	BD	80	F1	D0	DC	B7	80	9C	BE	B7	F3	BE	ED	80	00

HUNT THE WUMPUS (CONT.)

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0300	AD	06	17	85	C0	A9	FF	A2	0E	95	C1	CA	10	FB	A9	03
0310	85	E0	A0	05	10	02	A0	00	A2	05	20	72	02	29	0F	D5
0320	CA	F0	F5	CA	10	F9	99	CA	00	88	10	EC	20	B2	02	A0
0330	03	84	E1	B9	C6	00	20	8F	02	8A	30	17	E0	03	30	04
0340	A9	19	10	0A	E0	01	30	04	A9	0E	10	02	A9	00	A0	01
0350	20	00	02	C6	E1	A4	E1	10	DA	A4	CA	B9	E7	1F	85	0C
0360	A2	03	B4	C6	B9	E7	1F	95	20	CA	10	F6	A0	00	98	20
0370	00	02	20	58	02	C9	14	F0	48	20	C5	02	85	CA	8A	30
0380	EB	A5	CA	A2	04	D5	C1	F0	33	CA	10	F9	20	8F	02	8A
0390	30	9A	E0	03	10	17	E0	01	10	1D	A0	00	A9	26	20	00
03A0	02	20	99	02	C5	CA	D0	84	A9	26	20	CF	02	A0	01	A9
03B0	3D	20	00	02	4C	16	03	A9	4F	20	CF	02	A9	65	20	CF
03C0	02	A0	00	A9	B7	20	00	02	20	58	02	20	C5	02	85	D1
03D0	8A	30	EE	A5	D1	A6	E0	95	C0	C5	CB	F0	15	C6	E0	F0
03E0	1A	A4	E0	B9	E7	1F	85	9F	A0	00	A9	90	20	00	02	4C
03F0	6C	03	A0	02	A9	DE	20	00	02	F0	F7	A9	73	20	CF	02

APPENDIX E.

SPECIAL APPLICATIONS

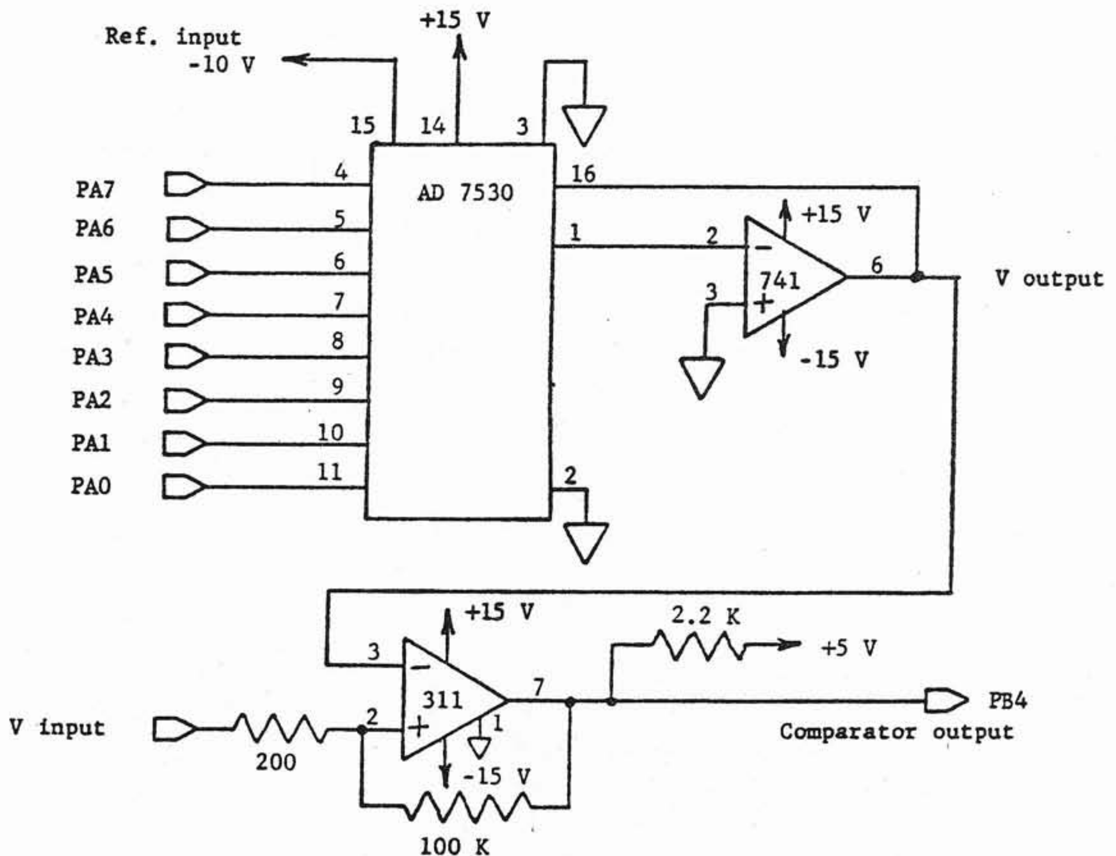
Single Channel Analog to Digital Conversion

SUBROUTINE ADC - 8 Bit Analog to Digital Conversion

```
0080 A9 80      ADC    LDA#  $80     enter trial no.
0082 85 EE             STAz  TRIAL   save it
0084 A9 00             LDA#  $00     clear A
0086 18         NXTBIT CLC           clear carry before add
0087 65 EE             ADCz  TRIAL   add trial to A
0089 8D 00 17          STA@  PAD     output to DAC
008C AD 02 17          LDA@  PBD     check comparator
008F 29 10             AND#  $10     mask all but bit 4
0091 D0 09             BNEr  SAVE    if comp. = 1, save result
0093 AD 00 17          LDA@  PAD     too big, get no. from DAC
0096 38                SEC           set carry before subtract
0097 E5 EE             SBCz  TRIAL   subtract trial no.
0099 4C 9F 00          JMP@  SHIFT
009C AD 00 17   SAVE   LDA@  PAD     load DAC into A
009F 46 EE      SHIFT  LSRz  TRIAL   divide trial by 2
00A1 90 E3             BCCr  NXTBIT  done if trial less than 1
00A3 60                RTS           return with final no. in A
```

Hardware:

PA0 - PA7 are out to DAC
PB4 is in from comparator



ANALOG TO DIGITAL CONVERSION DEMONSTRATION PROGRAM

Display ADC Output in HEX Format

```
0000 A9 FF      START   LDA#   $FF      set PA port to output
0002 8D 01 17           STA@   $1701
0005 AD 03 17           LDA@   $1703    set PB4 to be input
0008 29 EF              AND#   $EF
000A 8D 03 17           STA@   $1703
000D 20 80 00   LOOP    JSR@   ADC      call ADC subroutine
0010 85 F9              STAz   $F9      store ADC output in right display
0012 20 1F 1F           JSR@   SCANDS   display data
0015 4C 0D 00           JMP@   LOOP     loop back for more data
```

Display ADC Output in BCD Format

```
0020 A9 FF      START   LDA#   $FF      set PA port to output
0022 8D 01 17           STA@   PADD
0025 AD 03 17           LDA@   PBDD     set PB4 to be input
0028 29 EF              AND#   $EF
002A 8D 03 17           STA@   PBDD
002D 20 80 00   READ    JSR@   ADC      read ADC
0030 85 E7              STAz   HEDEC-L  set up data for binary to BCD conversio
0032 A2 00              LDX#   $00
0034 86 E6              STXz   HEDEC-H
0036 20 00 02           JSR@   HEDEC    call binary to BCD conversion routine
0039 A6 E1              LDXz   $E1      get BCD result high
003B 86 FB              STXz   $FB      store result in left display
003D A6 E2              LDXz   $E2      get BCD result low
003F 86 FA              STXz   $FA      store result in middle display
0041 A2 00              LDX#   $00      zero the right display
0043 86 F9              STXz   $F9
0045 20 1F 1F           JSR@   SCANDS   display final BCD value
0048 4C 2D 00           JMP@   READ     loop back for more data
```

note: In order to perform the binary to BCD conversion, you must load the HEDEC program into the memory starting at address $0200.

This program uses the circuit and ADC subroutine shown on page E-1.

MULTICHANNEL ANALOG INPUT/OUTPUT SYSTEM FOR KIM-1

by J.B. Ross

A multichannel analog I/O system which is ideally suited to the KIM-1 system was developed by Douglas R. Kraul (BYTE June 1977, pp. 18-23). This system (see diagram on p.E-7) provides 8 channels of analog input and output. The circuit uses standard components and can be constructed using wire-wrap techniques for less than $50.00.

The multichannel I/O system is interfaced to KIM-1 via the programmable I/O lines as follows:

Connect the 8 data lines driving the DAC to port PA--

```
DAC 0   to  PA0
DAC 1   to  PA1
  .      .    .
  .      .    .
DAC 7   to  PA7
```

Connect the remaining control lines to port PB--

```
SELECT 0   to  PB0
SELECT 1   to  PB1
SELECT 2   to  PB2
STROBE     to  PB3
SIGN BIT   to  PB4
```

The complete driver software is given on the following pages. The interface driver uses the KIM-1 interval timer to trigger an NMI interrupt to update the inputs and outputs every 50 mSec so you must also connect PB7 to pin 6 of the expansion connector. The NMI interrupt vector is set up by the initializing routine starting at $0380.

To make the interface system operate, load the Analog Interface Driver Routine, the Analog to Digital Conversion Subroutine, and the Initialization Routine. Start the program at $0380. Control will be transferred immediately back to the KIM monitor. If the display begins to flicker, the program is operating properly. Eight bit data to be sent to output channels 0 through 7 is stored in locations $00C0 through $00C7, eight bit input

data from channels 0 through 7 is written into locations \$00C8 through \$00CF. When the interface driver is operating, the keyboard monitor can be used to enter data for the analog outputs and to examine data from the analog inputs. This feature makes calibration of the interface very convenient.

Since the analog data is transferred to and from the memory table by the interface driver software, users need concern themselves only with the details of how to use the digital data contained in the table. Programs can readily be written to examine input data and generate output data. If a user program does any critical timing, there is a possibility of interference by the interface driver. A complete update of all inputs and outputs requires about 5 mSec and takes place automatically every 50 mSec. If a timming loop is used, it may be lengthened by 5 mSec. The interface driver uses the interval timer located at \$1704 - \$170F so this sould not be used by another program. The other timer at \$1744 - \$174F is available for general timing use, but has no signal output for interuppting the processor.

The driver software also includes an eight bit counter in location \$00D0 which increments by one each time the interface is serviced, and a four digit BCD "clock" in locations \$00D1 and \$00D2 which is incremented once each 0.1 Sec (approximately).

ANALOG INTERFACE DRIVER ROUTINE

```
0300 48          START   PHA             save A
0301 8A                  TXA
0302 48                  PHA             save X
0303 A2 00               LDX#   $00      clear X
0305 8A          OUTPUT  TXA             get channel number
0306 09 08               ORA#   $08      disable output multiplexer
0308 8D 02 17            STA@   $1702    select output channel
030B B5 C0               LDAzx  $C0      get number from memory table
030D 8D 00 17            STA@   $1700    send it to DAC
0310 AD 02 17            LDA@   $1702    enable output multiplexer
0313 29 F7               AND#   $F7
0315 8D 02 17            STA@   $1702
0318 A9 64       DELAY   LDA#   $64      set up time delay for charging
031A 8D 04 17            STA@   $1704    use microsecond timer
031D AD 07 17    WAIT    LDA@   $1707    get status
0320 F0 FB               BEQr   WAIT     wait until timer is done
0322 AD 02 17            LDA@   $1702    disable output multiplexer
0325 09 08               ORA#   $08
0327 8D 02 17            STA@   $1702
032A E8                  INX             increment channel in X
032B E0 08               CPX#   $08      check for maximum X=8
032D D0 D6               BNEr   OUTPUT   if less than maximum, repeat output
032F 8E 02 17    INPUT   STX@   $1702    select input channel
0332 20 58 03            JSR@   ADC      convert analog V to binary
0335 95 C0               STAzx  $C0      save number in memory table
0337 E8                  INX             increment channel in X
0338 E0 10               CPX#   $10      check for maximum X=$10
033A D0 F3               BNEr   INPUT    if less than maximum, repeat input
033C A9 30       EXIT    LDA#   $30      reload interval timer with refresh
033E 8D 0F 17            STA@   $170F        value (msec)
0341 E6 D0               INCz   COUNT    increment sample count
0343 18          CLOCK   CLC             clear carry before addition
0344 F8                  SED             switch to BCD mode
0345 A5 D0               LDAz   COUNT    get count number
0347 29 01               AND#   $01      mask all but low bit
0349 65 D1               ADCz   TIME-L   add bit to low order time
034B 85 D1               STAz   TIME-L   save result
034D A9 00               LDA#   $00      clear A
034F 65 D2               ADCz   TIME-H   add carry to high order time
0351 85 D2               STAz   TIME-H   save result
0353 D8                  CLD             return to binary mode
0354 68                  PLA             restore X
0355 AA                  TAX
0356 68                  PLA             restore A
0357 40                  RTI             return from interrupt
```

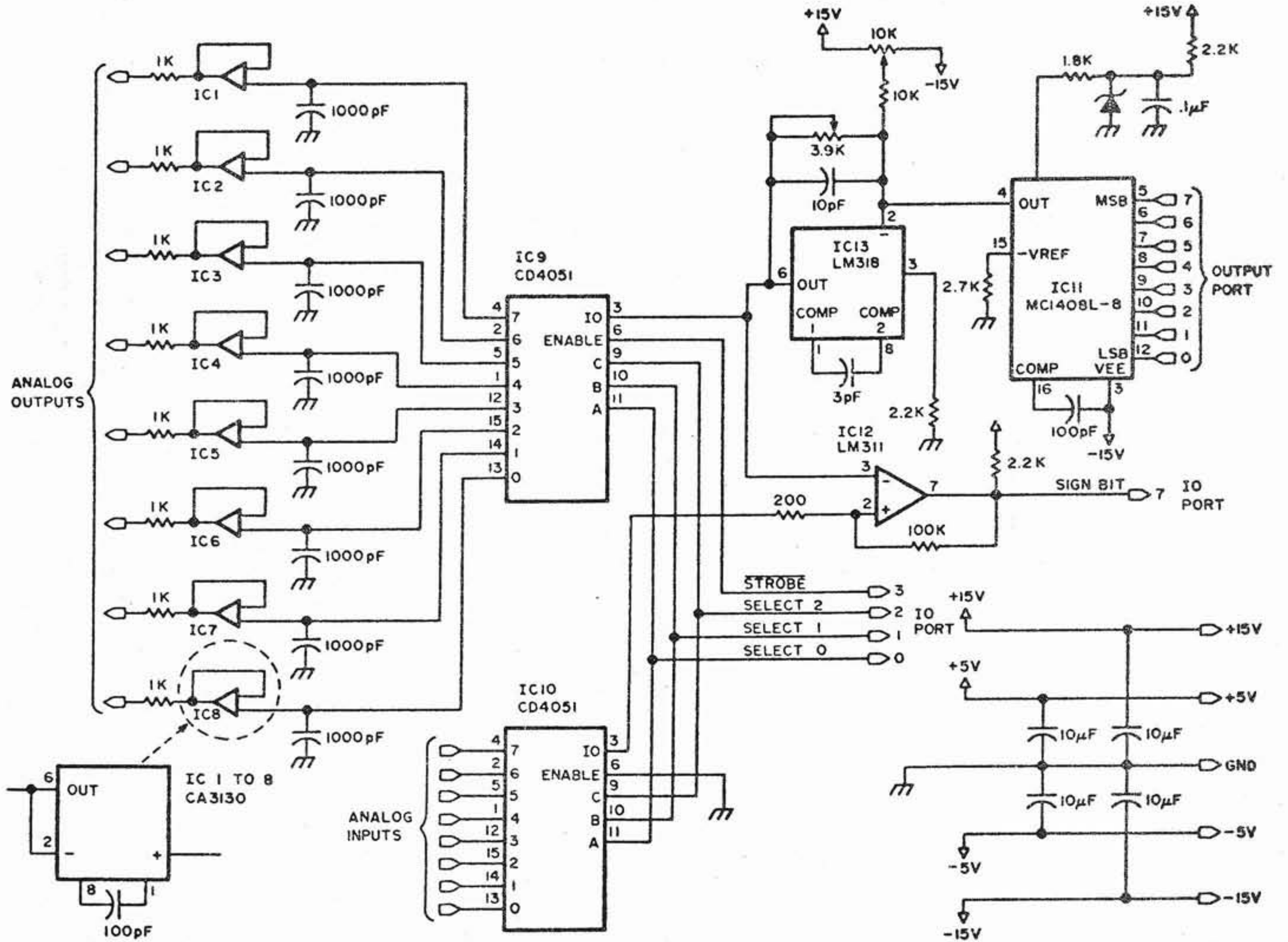
ANALOG TO DIGITAL CONVERSION SUBROUTINE

```
0358 A9 80      ADC     LDA#   $80      enter trial number
035A 85 EE              STAz   TRIAL    save it
035C A9 00              LDA#   $00      clear A
035E 18         NXTBIT  CLC             clear carry before addition
035F 65 EE              ADCz   TRIAL    add trial value to A
0361 8D 00 17           STA@   DAC      send trial value to DAC
0364 C6 F3              DECz   $F3      waste 5 microseconds
0366 AD 02 17           LDA@   COMP     get comparator status
0369 29 10              AND#   $10      mask to recover bit 4
036B D0 09              BNEr   SAVE     save result if comparator = 1
036D AD 00 17           LDA@   DAC      too big, get number from DAC
0370 38                 SEC             set carry before subtraction
0371 E5 EE              SBCz   TRIAL    subtract trial number
0373 18                 CLC
0374 90 03              BCCr   SHIFT    jump to shift
0376 AD 00 17   SAVE    LDA@   DAC      get number from DAC
0379 46 EE      SHIFT   LSRz   TRIAL    divide trial number by 2
037B 90 E1              BCCr   NXTBIT   done if carry is 1
037D 60                 RTS             return with final value in A
```

INITIALIZATION ROUTINE FOR INTERFACE DRIVER

```
0380 A9 FF      INTLZ   LDA#   $FF      set PA port to output
0382 8D 01 17           STA@   $1701
0385 A9 0F              LDA#   $0F      set PB0 - PB3 to output
0387 8D 03 17           STA@   $1703
038A A9 00              LDA#   $00      clear A
038C 85 D0              STAz   COUNT    clear COUNT
038E 85 D1              STAz   TIME-L   clear TIME-L
0390 85 D2              STAz   TIME-H   clear TIME-H
0392 8D FA 17           STA@   NMI-L    set up NMI interrupt vector
0395 A9 03              LDA#   $03
0397 8D FB 17           STA@   NMI-H
039A AD 0E 17           LDA@   $170E    enable timer interrupt
039D 4C 4F 1C           JMP@   $1C4F    jump to monitor (or user program)
```

MULTICHANNEL ANALOG INTERFACE



Number	Type	+5 V	GND	−15 V	+15 V	−5 V
1 to 8	CA3130	7				4
9	CD4051	16.	8			7
10	CD4051	16	8			7
11	MC1408L-8	13	2			
12	LM311		1	4	8	
13	LM318			4	8	

APPENDIX F.

KIM/6500 INFORMATION SOURCES

KIM SOFTWARE SOURCES

KIM-1/650X User Notes
109 Centre Ave.
West Norriton, PA 19401
Published every 5 to 8 weeks. Subscription: $5.00 for six issues. Back issues may be available. Highly recommended.

ARESCO
314 Second Ave.
Haddon Heights, NJ 08035
4K version of FOCAL for $40, 2.5K assembler (nonstandard mnemonics) for $30, 6K assembler/text editor (standard mnemonics) for $60. Send $2.00 for literature.

6502 PROGRAM EXCHANGE
2920 Moana
Reno, Nevada 89509
4K FOCAL (FCL-65), scientific routine package (written in FOCAL), games and general software for 6500 systems using the KIM and TIM monitors. Send $0.50 for program list.

THE COMPUTERIST
P.O. Box 3
S. Chelmsford, MA 01824
High quality software. PLEASE game package for KIM-1: $10.00 (cassette). HELP text editors and word processing programs-send for description-$15.00 per cassette. MICRO-CHESS Chess playing program for KIM-1: $15.00.

PYRAMID DATA SYSTEMS
6 Terrace Avenue
New Egypt, NJ 08533
"XIM" extended I/O monitor package for KIM (requires more than 1K of memory) $12 for manual and cassette.

MICRO-WARE LTD.
27 Firstbrooke Road
Toronto, Ontario
Canada M4E 2L2
Assembler, dissassembler, and text editor for 6502 with 4K memory. Manual and KIM cassette: $25, source listing: $25. Well documented.

Kenneth W. Ensele
1337 Foster Rd.
Napa, CA 94558
Source for Tom Pittmans 2K TINY BASIC on KIM cassette. Specify starting address $0200 or $2000. $9.50 for tape plus $1.00 handling and postage.

ORB
P.O. Box 311
Argonne, IL 60439

THE FIRST BOOK OF KIM by Stan Ockers, Jim Butterfield, and Eric Rehnke. The book includes a beginners guide to KIM, several tutorials on hooking things up to KIM, and a large number of game and utility type programs. 180 pages, 8 1/2 X 11 format: $9.00 plus $0.50 postage.

Johnson Computer
P.O. Box 523
Medina, Ohio 44258

4.5K assembler/text editor and other 6502 software. Write for current information.

6500 MICROPROCESSOR SUPPLIERS

MOS Technology
950 Rittenhouse Rd.
Norristown, PA 19401 (215) 666-7950

Rockwell Microelectronic Devices
P.O. Box 3669
Anaheim, CA 92803 (714) 632-3729

Synertek
3050 Coronado Dr.
Santa Clara, CA 95051 (408) 984-8900

6500 BASED MICROCOMPUTER SUPPLIERS

Apple Computer Inc.
20863 Stevens Creek Blvd.
Bldg. B3-C
Cupertino, CA 95014 (408) 996-1010

Commodore Business Machines
901 California Ave.
Palo Alto, CA 94304 (415) 326-4000

ECD Corp.
196 Broadway
Cambridge, MA 02139 (617) 661-4400

Ohio Scientific
11679 Hayden
Hiram, Ohio 44234

APPENDIX G

G. GENERAL REFERENCE INFORMATION

6530 TIMER FUNCTIONS AND PROPERTIES

A. TIME-OUT FLAG AND INTERRUPT ENABLE REGISTER:

1. ALL WRITE OPERATIONS TO THE COUNTER TOUCH THE INTERRUPT ENABLE REGISTER (ADDRESS BIT 3, THE '8' BIT, IS COPIED INTO THE INTERRUPT ENABLE REGISTER).

2. ALL READ OPERATIONS ON THE COUNTER (EVEN ADDRESSES) TOUCH THE INTERRUPT ENABLE REGISTER.

3. ALL READ OPERATIONS ON THE TIME-OUT FLAG (ODD ADDRESSES) LEAVE THE INTERRUPT ENABLE REGISTER UNTOUCHED.

4. AFTER COMPLETION OF TIME-OUT, FLAG READ OPERATIONS DO NOT CLEAR THE TIME-OUT FLAG.

5. AFTER COMPLETION OF TIME-OUT, COUNTER READ OPERATIONS CLEAR THE TIME-OUT FLAG.

6. ALL COUNTER WRITE OPERATIONS CLEAR THE TIME-OUT FLAG.

B. PRE-SCALER BITS:

1. PRE-SCALER BITS ARE TOUCHED ONLY BY WRITE OPERATIONS (ADDRESS BITS 0 AND 1, THE '1' AND '2' BITS, ARE COPIED INTO THE PRE-SCALER REGISTER).

2. THE COUNTER CAN BE LOADED AT ALL ADDRESSES FROM 1704-1707 AND FROM 170C-170F, BUT IT CAN BE READ ONLY AT THE EVEN ADDRESSES.

3. THE TIME-OUT FLAG CAN BE READ ONLY AT ODD ADDRESSES; SUCH READ OPERATIONS ALWAYS RETURN EITHER 00 OR 80 (HEX).

CONSEQUENCES:

1. SETTING THE PRE-SCALER BITS REQUIRES A WRITE OPERATION.

2. ENABLING THE INTERRUPT REQUIRES A WRITE OPERATION AT ADDRESSES 170C-170F, OR A READ OPERATION AT EITHER 170C OR 170E.

3. DISABLING THE INTERRUPT REQUIRES A WRITE OPERATION AT ADDRESSES 1704-1707, OR A READ OPERATION AT EITHER 1704 OR 1706.

4. ALL TRANSACTIONS AT EVEN ADDRESSES CLEAR THE TIME-OUT FLAG IF IT HAPPENED TO BE SET.

FD 11111

INTERRUPT EXPERIMENT

```
0000   A9  40       LDA# $40
0002   8D  FE  17   STA@ $17FE
0005   A9  00       LDA# $00
0007   8D  FF  17   STA@ $17FF     IRQ VECTOR INSTALLED
000A   8D  03  17   STA@ $1703     PORT B INPUT
000D   A9  FF       LDA# $FF
000F   8D  01  17   STA@ $1701     PORT A OUTPUT
0012   8D  0F  17   STA@ $170F     START UP TIMER   1024 time delay, interrupt ~1/4 sec
0015   58           CLI            ENABLE INTERRUPTS
0016   F8           SED            DECIMAL MODE
0017   A9  00       LDA# $00
0019   85  F9       STAZ $F9
001B   85  FA       STAZ $FA
001D   85  FB       STAZ $FB       ZERO OUT DISPLAY DIGITS
001F   38           STC            USE CARRY TO DO THE INCREMENT
0020   A2  FD       LDX# $FD       NOTE WRAP-AROUND INDEXING
0022   B5  FC       LDAZX $FC      TO GET TO LOCATION F9 FIRST
0024   69  00       ADC# $00
0026   95  FC       STAZX $FC      WRITE BACK UPDATED DIGIT PAIR
0028   90  03       BCC $03        FALL OUT IF NO CARRY-OUT
002A   E8           INX            UPDATE INDEX IF NEED BE
002B   D0  F5       BNE $F5        FALL OUT IF ALL DIGITS DONE
002D   A9  20       LDA# $20
002F   85  80       STAZ $80       USE LOC. 80 AS DISPLAY LOOP CTR.
0031   20  1F  1F   JSR@ $1F1F     CALL TO DISPLAY DIGITS
0034   C6  80       DECZ $80       COUNT DOWN DISPLAY CALLS
0036   D0  F9       BNE $F9        DO ANOTHER DISPLAY CALL
0038   F0  E5       BEQ $E5        UPDATE DISPLAY CONTENTS
```

NOW THE INTERRUPT-DRIVEN PROGRAM:

```
0040   48           PHA            SAVE ACCUMULATOR
0041   AD  02  17   LDA@ $1702     GET SWITCHES
0044   0A           ASL A          SHIFT UP
0045   0A           ASL A          TWICE
0046   D0  02       BNE $02        IF ALL SWITCHES ARE ZERO
0048   A9  FF       LDA# $FF       USE $FF FOR DEFAULT
004A   8D  0F  17   STA@ $170F     RESTART TIMER
004D   EE  00  17   INC@ $1700     UPDATE PORT A
0050   68           PLA            RETRIEVE ACCUMULATOR
0051   40           RTI            RETURN FROM INTERRUPT
```

NOTES:

1. GROUNDING SWITCHES WILL SPEED UP THE UPDATES ON PORT A.

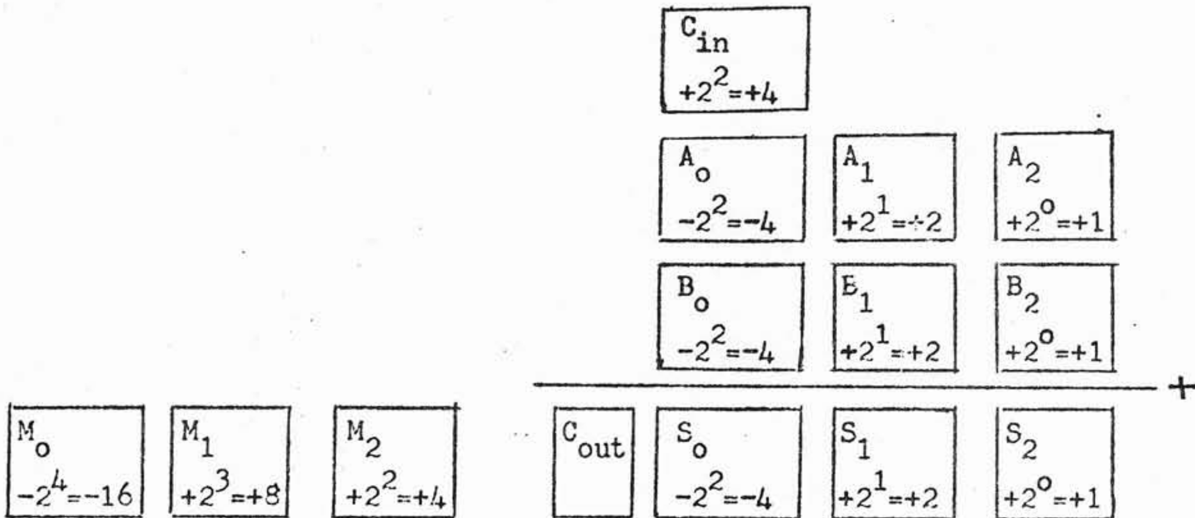
2. LOCATION 002E CONTROLS THE COUNTING RATE ON THE DIGIT DISPLAY.

CODE COMPARISON CHART

DECIMAL	4-BIT BINARY	SIGN AND MAGNITUDE (+ = 0 − = 1)	5-BIT OFFSET BINARY	1s COMPL. BINARY	2s COMP. BINARY	5-BIT GRAY
15	1111	0 1111	1 1111	0 1111	0 1111	01000
14	1110	0 1110	1 1110	0 1110	0 1110	01001
13	1101	0 1101	1 1101	0 1101	0 1101	01011
12	1100	0 1100	1 1100	0 1100	0 1100	01010
11	1011	0 1011	1 1011	0 1011	0 1011	01110
10	1010	0 1010	1 1010	0 1010	0 1010	01111
9	1001	0 1001	1 1001	0 1001	0 1001	01101
8	1000	0 1000	1 1000	0 1000	0 1000	01100
7	0111	0 0111	1 0111	0 0111	0 0111	00100
6	0110	0 0110	1 0110	0 0110	0 0110	00101
5	0101	0 0101	1 0101	0 0101	0 0101	00111
4	0100	0 0100	1 0100	0 0100	0 0100	00110
3	0011	0 0011	1 0011	0 0011	0 0011	00010
2	0010	0 0010	1 0010	0 0010	0 0010	00011
1	0001	0 0001	1 0001	0 0001	0 0001	00001
0	0000	0 0000	1 0000	0 0000 1 1111	0 0000	00000
−1		1 0001	0 1111	1 1110	1 1111	10000
−2		1 0010	0 1110	1 1101	1 1110	10001
−3		1 0011	0 1101	1 1100	1 1101	10011
−4		1 0100	0 1100	1 1011	1 1100	10010
−5		1 0101	0 1011	1 1010	1 1011	10110
−6		1 0110	0 1010	1 1001	1 1010	10111
−7		1 0111	0 1001	1 1000	1 1001	10101
−8		1 1000	0 1000	1 0111	1 1000	10100
−9		1 1001	0 0111	1 0110	1 0111	11100
−10		1 1010	0 0110	1 0101	1 0110	11101
−11		1 1011	0 0101	1 0100	1 0101	11111
−12		1 1100	0 0100	1 0011	1 0100	11110
−13		1 1101	0 0011	1 0010	1 0011	11010
−14		1 1110	0 0010	1 0001	1 0010	11011
−15		1 1111	0 0001	1 0000	1 0001	11001
−16			0 0000		1 0000	11000

Offset binary and 2s complement differ only in the state of the sign bit. Gray code is not weighted; it can only be converted into a binary bit string, which must then be further interpreted.

For the purpose at hand we will use a 3-bit binary adder which will accept a pair of 3-bit inputs (the addends) to form a 4-bit output by the rules of straight ("unsigned") binary addition. In addition to the 10 bits making up the inputs and the output, we will define one more signal, which is the carry into the leftmost bit position of the addends:



To keep a running count of overflow and underflow events we will need one more register, here also shown as 3 bits wide. Because spill-out from the adder may have a weight of +4 or -4 (overflow or underflow), it will be convenient to assign a weight of 4 to M_2, and to treat the contents of M as another signed number which may be incremented or decremented by the same add/subtract strategy we will develop for A, B, and S. The bit weights for all bits are shown in the diagram; they correspond to the standard convention for two's complement signed integers. In the examples below, the bit locations and formats will be as shown above, but the bit weights will not be shown.

Two factors may be held accountable for most of the confusion around signed arithmetic:

1. The term MSB (most significant bit) is often used with an implicit convention which may assign the name MSB to either bit 0 or bit 1 of a word. In recognition of this, we will not use the term MSB here.
2. The signals in the 0 column may have either a positive weight (C_{in}) or a negative weight (A_o, B_o). The adder makes no such distinction; the interpretation of bits as weighted numbers is strictly ours.

Having recognized the sources of confusion, let us attempt to create some order by inspecting all possible combinations of sign bits and carry-in signals:

A. $C_{in} = 0$; $A_o = 0$; $B_o = 0$

```
               0
               0 1 0    +2
               0 0 1    +1
          ------------  ---
0 0 0   0      0 1 1    +3
```

Addition of two small positive numbers.
No overflow, no underflow

B. $C_{in} = 0$; $A_o = 0$; $B_o = 1$ (or $A_o = 1$; $B_o = 0$)

```
               0
               0 1 0    +2
               1 0 1    -3
          ------------  ---
0 0 0   0      1 1 1    -1
```

Addition of a positive and a negative number, result negative.
No overflow, no underflow

C. $C_{in} = 0$; $A_o = 1$; $B_o = 1$

```
               0
               1 0 1    -3
               1 1 0    -2
          ------------  ---
0 0 0   1      0 1 1    -5
1 1 1
-----
1 1 1          1 1 1    -4 + -1
```

Addition of two negative numbers.
Underflow
C_o has a weight of 2 x -4, which is redistributed to M_2 and S_o

D. $C_{in} = 1$; $A_o = 0$; $B_o = 0$

```
               1
               0 1 1    +3
               0 0 1    +1
          ------------  ---
0 0 0   0      1 0 0    +4
0 0 1          0 0 0    +4 + 0
```

Addition of two positive numbers.
Overflow
S_o as formed has a weight of +4, which is moved to M_2

E. $C_{in} = 1$; $A_o = 0$; $B_o = 1$ (or $A_o = 1$; $B_o = 0$)

```
               1
               0 1 0    +2
               1 1 0    -2
          ------------  ---
0 0 0   1      0 0 0     0
```

Addition of a positive and a negative number.
No overflow, no underflow
Note that the weight of C_o is zero; C_o is produced by "addition" of C_{in}

with weight +4 and B_0 with weight -4.

F. $C_{in} = 1$; $A_0 = 1$; $B_0 = 1$

```
             1
             1 1 0     -2
             1 1 1     -1
        ----------    ----
0 0 0   1    1 0 1     -3
```

Addition of two small negative numbers.
No underflow, no overflow
Again C_{in} neutralizes one of the sign bits. The other sign bit reappears as S_0.

In summary:

1. If carry-in is produced, but no carry-out is generated, then overflow has occurred.
2. If a carry-out is produced without help from a carry-in, then underflow occurred.
3. If no carries are generated, neither overflow nor underflow occurred.
4. If a carry-in produces a carry-out, this amounts to neutralization of the positive weight of the carry-in by the negative weight of one of the sign bits. Neither overflow nor underflow has occurred.
5. Whenever overflow or underflow occurs, S_0 must be complemented, to make S suitable as input to the adder for further arithmetic operations. If $C_{out}=1$, M must be decremented; if $C_{out}=0$, M must be incremented.

In logical terms, spill-out can be detected as $C_{in} \oplus C_{out}$. The sign of the spill-out is given by C_{out}.

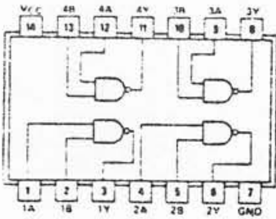
When all additions have been made, we should combine the spill-out counter with the S register to make a double-length bit string representing the end result in two's complement format. Now there are two bit positions with an absolute weight of 4: the low-order sign bit, S_0, and the LSB of the spill-out counter, M_2. These two bits must be combined, and then the vacated bit position must be eliminated to shift the high order bits into positions corresponding to their assigned weight.

APPENDIX H

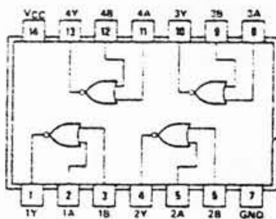
TTL REFERENCE SHEETS

TTL REFERENCE SHEET #1

7400



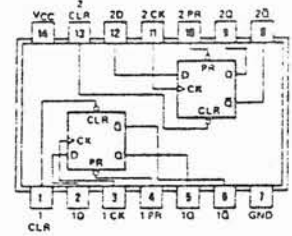
7402



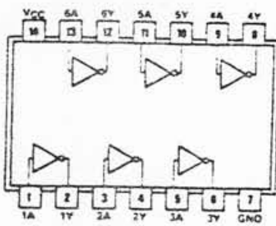
7474

FUNCTION TABLE

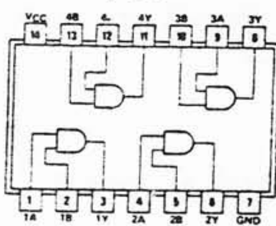
INPUTS				OUTPUTS	
PRESET	CLEAR	CLOCK	D	Q	$\bar{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H*	H*
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\bar{Q}_0$



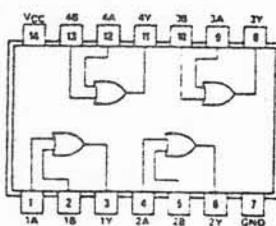
7404



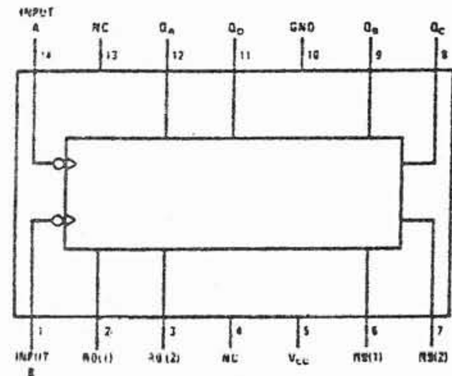
7408



7432



7490 DECADE COUNTER



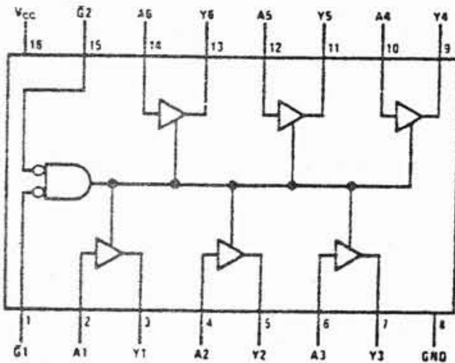
BCD COUNT SEQUENCE
(See Note A)

COUNT	OUTPUT			
	Q_D	Q_C	Q_B	Q_A
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	L	H	L	H
6	L	H	H	L
7	L	H	H	H
8	H	L	L	L
9	H	L	L	H

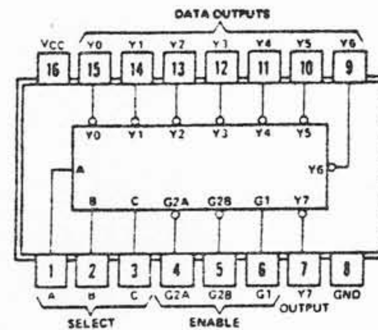
RESET/COUNT TRUTH TABLE

RESET INPUTS				OUTPUT			
R0(1)	R0(2)	R9(1)	R9(2)	Q_D	Q_C	Q_B	Q_A
H	H	L	X	L	L	L	L
H	H	X	L	L	L	L	L
X	X	H	H	H	L	L	H
X	L	X	L	COUNT			
L	X	L	X	COUNT			
L	X	X	L	COUNT			
X	L	L	X	COUNT			

8095 TRI-STATE BUFFER



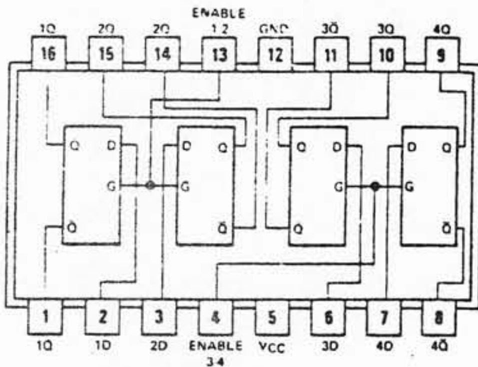
74138 1-OF-8 DECODER



INPUTS					OUTPUTS							
ENABLE		SELECT										
G1	G2*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

*G2 = G2A + G2B

7475 QUAD LATCH



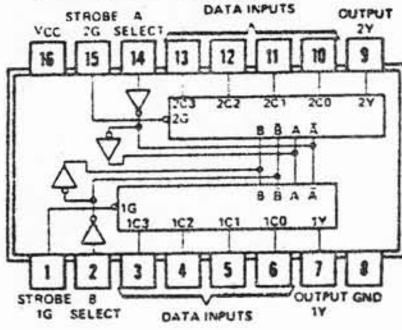
FUNCTION TABLE
(Each Latch)

INPUTS		OUTPUTS	
D	G	Q	$\bar{Q}$
L	H	L	H
H	H	H	L
X	L	Q_0	$\bar{Q}_0$

H = high level, L = low level, X = irrelevant

Q_0 = the level of Q before the high-to-low transition of G

74153 MULTIPLEXER

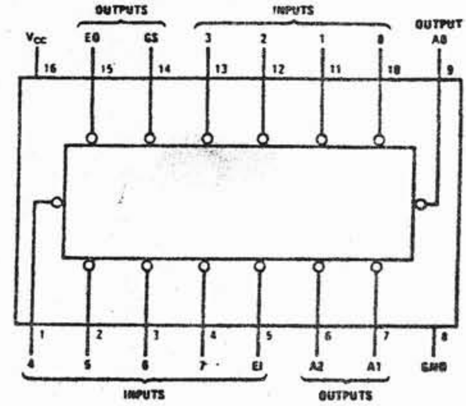


FUNCTION TABLE

SELECT INPUTS		DATA INPUTS				STROBE	OUTPUT
B	A	C0	C1	C2	C3	G	Y
X	X	X	X	X	X	H	L
L	L	L	X	X	X	L	L
L	L	H	X	X	X	L	H
L	H	X	L	X	X	L	L
L	H	X	H	X	X	L	H
H	L	X	X	L	X	L	L
H	L	X	X	H	X	L	H
H	H	X	X	X	L	L	L
H	H	X	X	X	H	L	H

Select inputs A and B are common to both sections.
H = high level, L = low level, X = irrelevant

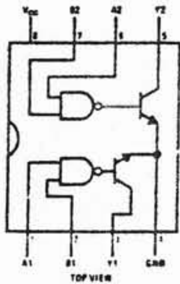
74148 PRIORITY ENCODER



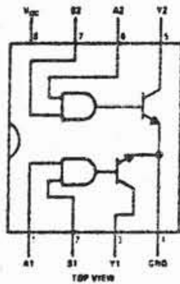
54148/74148

INPUTS									OUTPUTS				
EI	0	1	2	3	4	5	6	7	A2	A1	A0	GS	EO
H	X	X	X	X	X	X	X	X	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	L
L	X	X	X	X	X	X	X	L	L	L	L	L	H
L	X	X	X	X	X	X	L	H	L	L	H	L	H
L	X	X	X	X	X	L	H	H	L	H	L	L	H
L	X	X	X	X	L	H	H	H	L	H	H	L	H
L	X	X	X	L	H	H	H	H	H	L	L	L	H
L	X	X	L	H	H	H	H	H	H	L	H	L	H
L	X	L	H	H	H	H	H	H	H	H	L	L	H
L	L	H	H	H	H	H	H	H	H	H	H	L	H

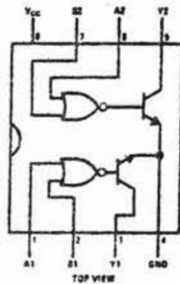
DUAL PERIPHERAL DRIVERS 300 mA, 20 V



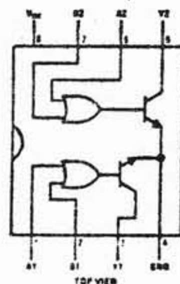
75451



75452



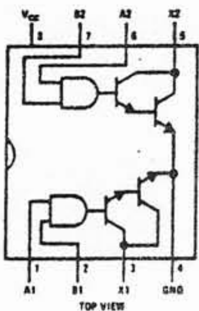
75453



75454

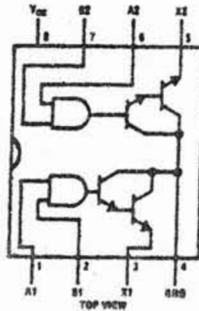
RELAY DRIVERS

+V



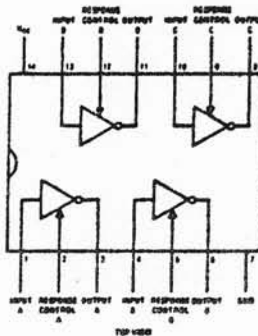
DS3686

-V

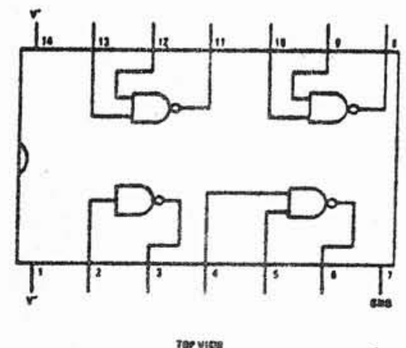


DS3687

RS-232C - TTL CONVERTERS



1489



1488

APPENDIX I

MOS TECHNOLOGY SHEETS

MOS TECHNOLOGY, INC.
VALLEY FORGE CORPORATE CENTER (215) 666-7950
950 RITTENHOUSE ROAD, NORRISTOWN, PA. 19401

PRELIMINARY

DATA

SHEET

MAY, 1976

MCS6500 MICROPROCESSORS

The MCS6500 Microprocessor Family Concept ----

The MCS6500 Series Microprocessors represent the first totally software compatible microprocessor family. This family of products includes a range of software compatible microprocessors which provide a selection of addressable memory range, interrupt input options and on-chip clock osscillators and drivers. All of the microprocessors in the MCS6500 group are software compatible within the group and are bus compatible with the M6800 product offering.

The family includes five microprocessors with on-board clock oscillators and drivers and four microprocessors driven by external clocks. The on-chip clock versions are aimed at high performance, low cost applications where single phase inputs, crystal or RC inputs provide the time base. The external clock versions are geared for the multi processor system applications where maximum timing control is mandatory. All versions of the microprocessors are available in 1 MHz and 2 MHz ("A" suffix on product numbers) maximum operating frequencies.

Features of the MCS6500 Family

- Single five volt supply
- N channel, silicon gate, depletion load technology
- Eight bit parallel processing
- 56 Instructions
- Decimal and binary arithmetic
- Thirteen addressing modes
- True indexing capability
- Programmable stack pointer
- Variable length stack
- Interrupt capability
- Non-maskable interrupt
- Use with any type or speed memory
- Bi-directional Data Bus
- Instruction decoding and control
- Addressable memory range of up to 65K bytes
- "Ready" input
- Direct memory access capability
- Bus compatible with MC6800
- Choice of external or on-board clocks
- 1MHz and 2MHz operation
- On-the-chip clock options
  - * External single clock input
  - * RC time base input
  - * Crystal time base input
- 40 and 28 pin package versions
- Pipeline architecture

Members of the Family

Microprocessors with On-Board Clock Oscillator

- MCS6502
- MCS6503
- MCS6504
- MCS6505
- MCS6506

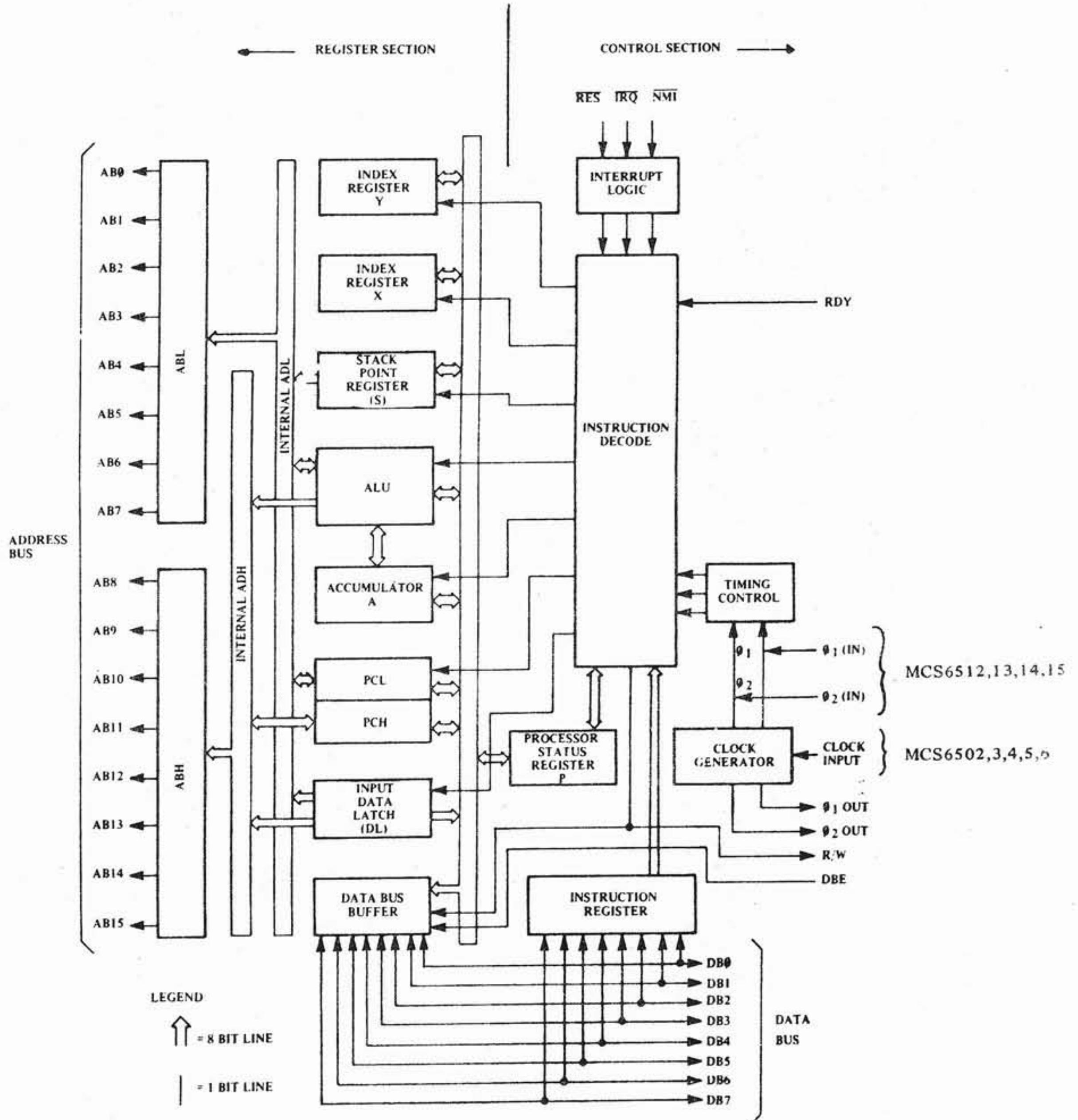
Microprocessors with External Two Phase Clock Input

- MCS6512
- MCS6513
- MCS6514
- MCS6515

Comments on the Data Sheet

The data sheet is constructed to review first the basic "Common Characteristics" - those features which are common to the general family of microprocessors. Subsequent to a review of the family characteristics will be sections devoted to each member of the group with specific features of each.

COMMON CHARACTERISTICS



Note: 1. Clock Generator is not included on MCS6512,13,14,15

2. Addressing Capability and control options vary with each of the MCS6500 Products.

MCS6500 Internal Architecture

INSTRUCTION SET – ALPHABETIC SEQUENCE

ADC Add Memory to Accumulator with Carry
AND "AND" Memory with Accumulator
ASL Shift left One Bit (Memory or Accumulator)

BCC Branch on Carry Clear
BCS Branch on Carry Set
BEQ Branch on Result Zero
BIT Test Bits in Memory with Accumulator
BMI Branch on Result Minus
BNE Branch on Result not Zero
BPL Branch on Result Plus
BRK Force Break
BVC Branch on Overflow Clear
BVS Branch on Overflow Set

CLC Clear Carry Flag
CLD Clear Decimal Mode
CLI Clear Interrupt Disable Bit
CLV Clear Overflow Flag
CMP Compare Memory and Accumulator
CPX Compare Memory and Index X
CPY Compare Memory and Index Y

DEC Decrement Memory by One
DEX Decrement Index X by One
DEY Decrement Index Y by One

EOR "Exclusive-or" Memory with Accumulator

INC Increment Memory by One
INX Increment Index X by One
INY Increment Index Y by One

JMP Jump to New Location
JSR Jump to New Location Saving Return Address

LDA Load Accumulator with Memory
LDX Load Index X with Memory
LDY Load Index Y with Memory
LSR Shift One Bit Right (Memory or Accumulator)

NOP No Operation

ORA "OR Memory with Accumulator

PHA Push Accumulator on Stack
PHP Push Processor Status on Stack
PLA Pull Accumulator from Stack
PLP Pull Processor Status from Stack

ROL Rotate One Bit Left (Memory or Accumulator)
ROR Rotate One Bit Right (Memory or Accumulator)
RTI Return from Interrupt
RTS Return from Subroutine

SBC Subtract Memory from Accumulator with Borrow
SEC Set Carry Flag
SED Set Decimal Mode
SEI Set Interrupt Disable Status
STA Store Accumulator in Memory
STX Store Index X in Memory
STY Store Index Y in Memory

TAX Transfer Accumulator to Index X
TAY Transfer Accumulator to Index Y
TSX Transfer Stack Pointer to Index X
TXA Transfer Index X to Accumulator
TXS Transfer Index X to Stack Pointer
TYA Transfer Index Y to Accumulator

ADDRESSING MODES

ACCUMULATOR ADDRESSING - This form of addressing is represented with a one byte instruction, implying an operation on the accumulator.

IMMEDIATE ADDRESSING - In immediate addressing, the operand is contained in the second byte of the instruction, with no further memory addressing required.

ABSOLUTE ADDRESSING - In absolute addressing, the second byte of the instruction specifies the eight low order bits of the effective address while the third byte specifies the eight high order bits. Thus, the absolute addressing mode allows access to the entire 65K bytes of addressable memory.

ZERO PAGE ADDRESSING - The zero page instructions allow for shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high address byte. Careful use of the zero page can result in significant increase in code efficiency.

INDEXED ZERO PAGE ADDRESSING - (X, Y indexing) - This form of addressing is used in conjunction with the index register and is referred to as "Zero Page, X" or "Zero Page, Y". The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally due to the "Zero Page" addressing nature of this mode, no carry is added to the high order 8 bits of memory and crossing of page boundaries does not occur.

INDEXED ABSOLUTE ADDRESSING - (X, Y indexing) - This form of addressing is used in conjunction with X and Y index register and is referred to as "Absolute, X", and "Absolute, Y". The effective address is formed by adding the contents of X or Y to the address contained in the second and third bytes of the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

IMPLIED ADDRESSING - In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

RELATIVE ADDRESSING - Relative addressing is used only with branch instructions and establishes a destination for the conditional branch.

The second byte of the instruction becomes the operand which is an "Offset" added to the contents of the lower eight bits of the program counter when the counter is set at the next instruction. The range of the offset is -128 to +127 bytes from the next instruction.

INDEXED INDIRECT ADDRESSING - In indexed indirect addressing (referred to as (Indirect,X)), the second byte of the instruction is added to the contents of the X index register, discarding the carry. The result of this addition points to a memory location on page zero whose contents is the low order eight bits of the effective address. The next memory location in page zero contains the high order eight bits of the effective address. Both memory locations specifying the high and low order bytes of the effective address must be in page zero.

INDIRECT INDEXED ADDRESSING - In indirect indexed addressing (referred to as (Indirect),Y), the second byte of the instruction points to a memory location in page zero. The contents of this memory location is added to the contents of the Y index register, the result being the low order eight bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high order eight bits of the effective address.

ABSOLUTE INDIRECT - The second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits of that memory location is contained in the third byte of the instruction. The contents of the fully specified memory location is the low order byte of the effective address. The next memory location contains the high order byte of the effective address which is loaded into the sixteen bits of the program counter.

PROGRAMMING MODEL

Bits	Register	Name	Symbol
7 – 0	A	ACCUMULATOR	A
7 – 0	Y	INDEX REGISTER	Y
7 – 0	X	INDEX REGISTER	X
15 – 8 / 7 – 0	PCH / PCL	PROGRAM COUNTER	"PC"
8 / 7 – 0	1 / S	STACK POINTER	"S"

PROCESSOR STATUS REG "P" (bits 7 – 0): N V _ B D I Z C

- C — CARRY 1 = TRUE
- Z — ZERO 1 = RESULT ZERO
- I — IRQ DISABLE 1 = DISABLE
- D — DECIMAL MODE 1 = TRUE
- B — BRK COMMAND
- V — OVERFLOW 1 = TRUE
- N — NEGATIVE 1 = NEG.

INSTRUCTION SET – OP CODES, Execution Time, Memory Requirements

INSTRUCTIONS: MNEMONIC	OPERATION	IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM.			IMPLIED			(IND,X)			(IND),Y			Z,PAGE,X			ABS,X			ABS,Y			RELATIVE			INDIRECT			Z,PAGE,Y			CONDITION CODES
		OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	N Z C I D V
ADC	A+M+C → A (4)(1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										✓ ✓ ✓ - - ✓
AND	A∧M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										✓ ✓ - - - -
ASL	C ← ▭ ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3													✓ ✓ ✓ - - -
BCC	BRANCH ON C=0 (2)																															90	2	2							- - - - - -
BCS	BRANCH ON C=1 (2)																															B0	2	2							- - - - - -
BEQ	BRANCH ON Z=1 (2)																															F0	2	2							- - - - - -
BIT	A∧M				2C	4	3	24	3	2																															M_7 ✓ - - - M_6
BMI	BRANCH ON N=1 (2)																															30	2	2							- - - - - -
BNE	BRANCH ON Z=0 (2)																															D0	2	2							- - - - - -
BPL	BRANCH ON N=0 (2)																															10	2	2							- - - - - -
BRK	(See Fig. 1)													00	7	1																									- - - 1 - -
BVC	BRANCH ON V=0 (2)																															50	2	2							- - - - - -
BVS	BRANCH ON V=1 (2)																															70	2	2							- - - - - -
CLC	0 → C													18	2	1																									- - 0 - - -
CLD	0 → D													D8	2	1																									- - - - 0 -
CLI	0 → I													58	2	1																									- - - 0 - -
CLV	0 → V													B8	2	1																									- - - - - 0
CMP	A − M (1)	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										✓ ✓ ✓ - - -
CPX	X − M	E0	2	2	EC	4	3	E4	3	2																															✓ ✓ ✓ - - -
CPY	Y − M	C0	2	2	CC	4	3	C4	3	2																															✓ ✓ ✓ - - -
DEC	M − 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													✓ ✓ - - - -
DEX	X − 1 → X													CA																											✓ ✓ - - - -
DEY	Y − 1 → Y													88																											✓ ✓ - - - -
EOR	A⊻M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										✓ ✓ - - - -
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													✓ ✓ - - - -
INX	X + 1 → X													E8	2	1																									✓ ✓ - - - -
INY	Y + 1 → Y													C8	2	1																									✓ ✓ - - - -
JMP	JUMP TO NEW LOC				4C	3	3																												6C	5	3				- - - - - -
JSR	(See Fig. 2) JUMP SUB				20	6	3																																		- - - - - -
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										✓ ✓ - - - -

MNEMONIC	OPERATION	IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM.			IMPLIED			(IND,X)			(IND),Y			Z,PAGE,X			ABS,X			ABS,Y			RELATIVE			INDIRECT			Z,PAGE,Y			CONDITION CODES
		OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	N Z C I D V
LDX	M → X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2	✓ ✓ - - - -
LDY	M → Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3													✓ ✓ - - - -
LSR	0 → ▭ → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3													0 ✓ ✓ - - -
NOP	NO OPERATION													EA	2	1																									- - - - - -
ORA	A∨M → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										✓ ✓ - - - -
PHA	A → M_s S − 1 → S													48	3	1																									- - - - - -
PHP	P → M_s S − 1 → S													08	3	1																									- - - - - -
PLA	S + 1 → S M_s → A													68	4	1																									✓ ✓ - - - -
PLP	S + 1 → S M_s → P													28	4	1																									(RESTORED)
ROL	▭ ← C (rotate left)				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3													✓ ✓ ✓ - - -
ROR	C → ▭ (rotate right)				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3													✓ ✓ ✓ - - -
RTI	(See Fig. 1) RTRN INT													40	6	1																									(RESTORED)
RTS	(See Fig. 2) RTRN SUB													60	6	1																									- - - - - -
SBC	A − M − C̄ → A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										✓ ✓ (3) - - ✓
SEC	1 → C													38	2	1																									- - 1 - - -
SED	1 → D													F8	2	1																									- - - - 1 -
SEI	1 → I													78	2	1																									- - - 1 - -
STA	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										
STX	X → M				8E	4	3	86	3	2																												96	4	2	
STY	Y → M				8C	4	3	84	3	2													94	4	2																
TAX	A → X													AA	2	1																									✓ ✓ - - - -
TAY	A → Y													A8	2	1																									✓ ✓ - - - -
TSX	S → X													BA	2	1																									✓ ✓ - - - -
TXA	X → A													8A	2	1																									✓ ✓ - - - -
TXS	X → S													9A	2	1																									
TYA	Y → A													98	2	1																									✓ ✓ - - - -

(1) ADD 1 TO "N" IF PAGE BOUNDARY IS CROSSED

(2) ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE
ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE

(3) CARRY NOT = BORROW

(4) IF IN DECIMAL MODE Z FLAG IS INVALID
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT

X INDEX X
Y INDEX Y
A ACCUMULATOR
M MEMORY PER EFFECTIVE ADDRESS
M_s MEMORY PER STACK POINTER
\+ ADD
− SUBTRACT
∧ AND
∨ OR
⊻ EXCLUSIVE OR
✓ MODIFIED
\- NOT MODIFIED
M_7 MEMORY BIT 7
M_6 MEMORY BIT 6
N NO. CYCLES
\# NO. BYTES

MCS6502 – 40 Pin Package

Signal	Pin	Pin	Signal
Vss	1	40	$\overline{RES}$
RDY	2	39	Ø2(OUT)
Ø1(OUT)	3	38	S.O.
$\overline{IRQ}$	4	37	Ø0(IN)
N.C.	5	36	N.C.
$\overline{NMI}$	6	35	N.C.
SYNC	7	34	R/W
Vcc	8	33	DB0
AB0	9	32	DB1
AB1	10	31	DB2
AB2	11	30	DB3
AB3	12	29	DB4
AB4	13	28	DB5
AB5	14	27	DB6
AB6	15	26	DB7
AB7	16	25	AB15
AB8	17	24	AB14
AB9	18	23	AB13
AB10	19	22	AB12
AB11	20	21	Vss

MCS6502

* 65K Addressable Bytes of Memory
* $\overline{IRQ}$ Interrupt
* $\overline{NMI}$ Interrupt
* On-the-chip Clock
  - ✓ TTL Level Single Phase Input
  - ✓ RC Time Base Input
  - ✓ Crystal Time Base Input
* SYNC Signal
  (can be used for single instruction execution)
* RDY Signal
  (can be used for single cycle execution)
* Two Phase Output Clock for Timing of Support Chips

Features of MCS6502

MCS6503 – 28 Pin Package

Signal	Pin	Pin	Signal
$\overline{RES}$	1	28	Ø2(OUT)
Vss	2	27	Ø0(IN)
$\overline{IRQ}$	3	26	R/W
$\overline{NMI}$	4	25	DB0
Vcc	5	24	DB1
AB0	6	23	DB2
AB1	7	22	DB3
AB2	8	21	DB4
AB3	9	20	DB5
AB4	10	19	DB6
AB5	11	18	DB7
AB6	12	17	AB11
AB7	13	16	AB10
AB8	14	15	AB9

MCS6503

* 4K Addressable Bytes of Memory (AB00-AB11)
* On-the-chip Clock
* $\overline{IRQ}$ Interrupt
* $\overline{NMI}$ Interrupt
* 8 Bit Bi-Directional Data Bus

Features of MCS6503

MCS6504 – 28 Pin Package

Signal	Pin	Pin	Signal
$\overline{RES}$	1	28	Ø2(OUT)
Vss	2	27	Ø0(IN)
$\overline{IRQ}$	3	26	R/W
Vcc	4	25	DB0
AB0	5	24	DB1
AB1	6	23	DB2
AB2	7	22	DB3
AB3	8	21	DB4
AB4	9	20	DB5
AB5	10	19	DB6
AB6	11	18	DB7
AB7	12	17	AB12
AB8	13	16	AB11
AB9	14	15	AB10

MCS6504

* 8K Addressable Bytes of Memory (AB00-AB12)
* On-the-chip Clock
* $\overline{IRQ}$ Interrupt
* 8 Bit Bi-Directional Data Bus

Features of MCS6504

MCS6505 – 28 Pin Package

Pin	Signal	Pin	Signal
1	$\overline{RES}$	28	Ø2(OUT)
2	Vss	27	Ø0(IN)
3	RDY	26	R/W
4	$\overline{IRQ}$	25	DB0
5	Vcc	24	DB1
6	AB0	23	DB2
7	AB1	22	DB3
8	AB2	21	DB4
9	AB3	20	DB5
10	AB4	19	DB6
11	AB5	18	DB7
12	AB6	17	AB11
13	AB7	16	AB10
14	AB8	15	AB9

MCS6505

* 4K Addressable Bytes of Memory (AB00-AB11)
* On-the-chip Clock
* $\overline{IRQ}$ Interrupt
* RDY Signal
* 8 Bit Bi-Directional Data Bus

Features of MCS6505

MCS6506 – 28 Pin Package

Pin	Signal	Pin	Signal
1	$\overline{RES}$	28	Ø2(OUT)
2	Vss	27	Ø0(IN)
3	Ø1(OUT)	26	R/W
4	$\overline{IRQ}$	25	DB0
5	Vcc	24	DB1
6	AB0	23	DB2
7	AB1	22	DB3
8	AB2	21	DB4
9	AB3	20	DB5
10	AB4	19	DB6
11	AB5	18	DB7
12	AB6	17	AB11
13	AB7	16	AB10
14	AB8	15	AB9

MCS6506

* 4K Addressable Bytes of Memory (AB00-AB11)
* On-the-chip Clock
* $\overline{IRQ}$ Interrupt
* Two phases off
* 8 Bit Bi-Directional Data Bus

Features of MCS6506

MCS6512 – 40 Pin Package

Pin	Signal	Pin	Signal
1	Vss	40	$\overline{RES}$
2	RDY	39	Ø2(OUT)
3	Ø1	38	S.O.
4	$\overline{IRQ}$	37	Ø2
5	Vss	36	DBE
6	$\overline{NMI}$	35	N.C.
7	SYNC	34	R/W
8	Vcc	33	DB0
9	AB0	32	DB1
10	AB1	31	DB2
11	AB2	30	DB3
12	AB3	29	DB4
13	AB4	28	DB5
14	AB5	27	DB6
15	AB6	26	DB7
16	AB7	25	AB15
17	AB8	24	AB14
18	AB9	23	AB13
19	AB10	22	AB12
20	AB11	21	Vss

MCS6512

* 65K Addressable Bytes of Memory
* $\overline{IRQ}$ Interrupt
* $\overline{NMI}$ Interrupt
* RDY Signal
* 8 Bit Bi-Directional Data Bus
* SYNC Signal
* Two phase input
* Data Bus Enable

Features of MCS6512

MCS6513 – 28 Pin Package

Signal	Pin	Pin	Signal
Vss	1	28	$\overline{RES}$
Ø1	2	27	Ø2
$\overline{IRQ}$	3	26	R/W
$\overline{NMI}$	4	25	DB0
Vcc	5	24	DB1
AB0	6	23	DB2
AB1	7	22	DB3
AB2	8	21	DB4
AB3	9	20	DB5
AB4	10	19	DB6
AB5	11	18	DB7
AB6	12	17	AB11
AB7	13	16	AB10
AB8	14	15	AB9

MCS6513

* 4K Addressable Bytes of Memory (AB00-AB11)
* Two phase clock input
* $\overline{IRQ}$ Interrupt
* $\overline{NMI}$ Interrupt
* 8 Bit Bi-Directional Data Bus

Features of MCS6513

MCS6514 – 28 Pin Package

Signal	Pin	Pin	Signal
Vss	1	28	$\overline{RES}$
Ø1	2	27	Ø2
$\overline{IRQ}$	3	26	R/W
Vcc	4	25	DB0
AB0	5	24	DB1
AB1	6	23	DB2
AB2	7	22	DB3
AB3	8	21	DB4
AB4	9	20	DB5
AB5	10	19	DB6
AB6	11	18	DB7
AB7	12	17	AB12
AB8	13	16	AB11
AB9	14	15	AB10

MCS6514

* 8K Addressable Bytes of Memory (AB00-AB12)
* Two phase clock input
* $\overline{IRQ}$ Interrupt
* 8 Bit Bi-Directional Data Bus

Features of MCS6514

MCS6515 – 28 Pin Package

Signal	Pin	Pin	Signal
Vss	1	28	$\overline{RES}$
RDY	2	27	Ø2
Ø1	3	26	R/W
$\overline{IRQ}$	4	25	DB0
Vcc	5	24	DB1
AB0	6	23	DB2
AB1	7	22	DB3
AB2	8	21	DB4
AB3	9	20	DB5
AB4	10	19	DB6
AB5	11	18	DB7
AB6	12	17	AB11
AB7	13	16	AB10
AB8	14	15	AB9

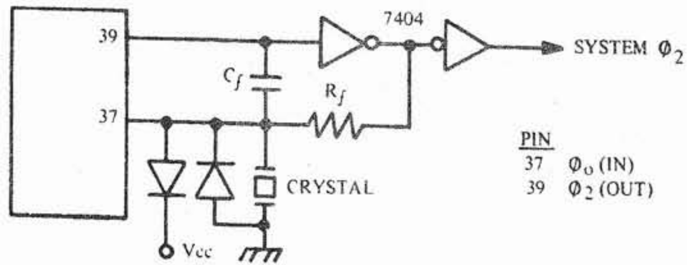
MCS6515

* 4K Addressable Bytes of Memory (AB00-AB11)
* Two phase clock input
* $\overline{IRQ}$ Interrupt
* 8 Bit Bi-Directional Data Bus

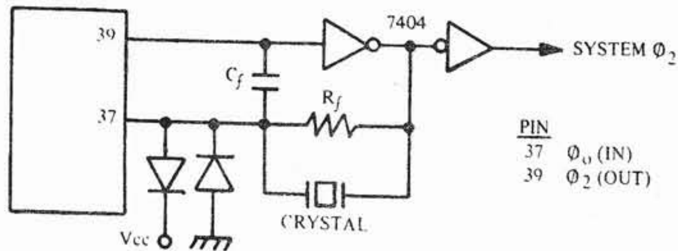
Features of MCS6515

TIME BASE GENERATION OF INPUT CLOCK

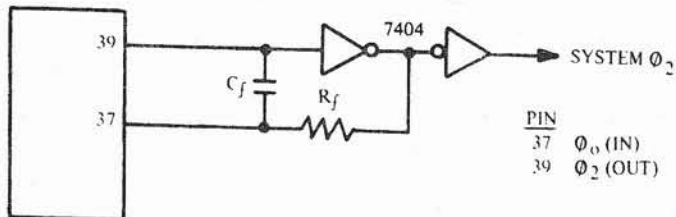
MCS6502



MCS6502 Parallel Mode Crystal Controlled Oscillator

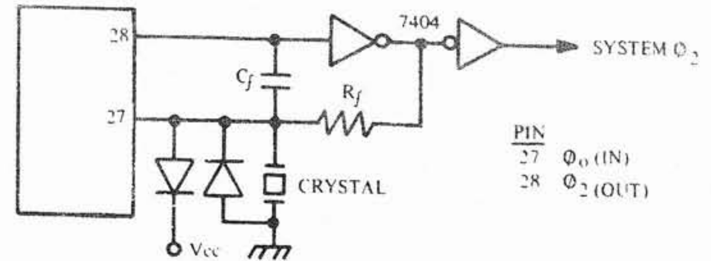


MCS6502 Series Mode Crystal Controlled Oscillator

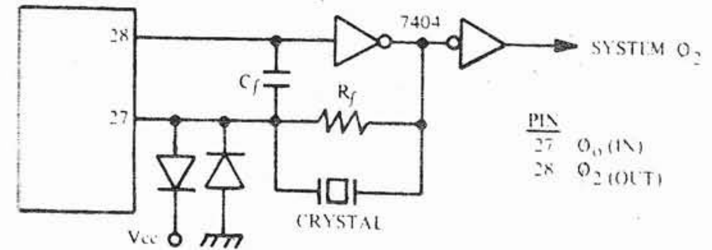


MCS6502 Time Base Generator – RC Network

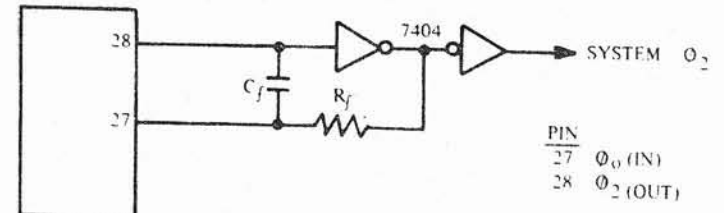
MCS6503, MCS6504, MCS6505, MCS6506



MCS6503,4,5,6 Parallel Mode Crystal Controlled Oscillator



MCS6503,4,5,6 Series Mode Crystal Controlled Oscillator



MCS6503, MCS6504, MCS6505, MCS6506 Time Base Generation RC Network

PRODUCT ANNOUNCEMENT BULLETIN

SEPTEMBER, 1976

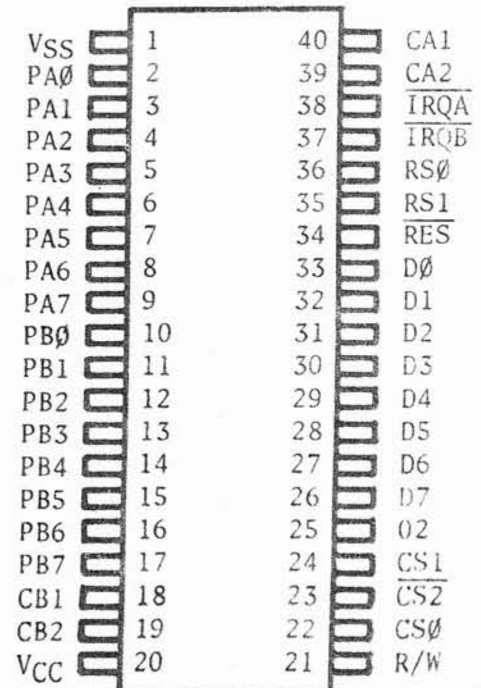
MCS6520 PERIPHERAL ADAPTER

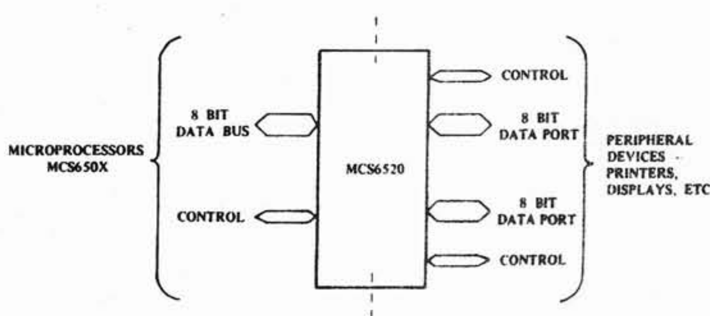
DESCRIPTION

The MCS6520 Peripheral Adapter is designed to solve a broad range of peripheral control problems in the implementation of microcomputer systems. This device allows a very effective trade-off between software and hardware by providing significant capability and flexibility in a low cost chip. When coupled with the power and speed of the MCS6500 family of microprocessors, the MCS6520 allows implementation of very complex systems at a minimum overall cost.

Control of peripheral devices is handled primarily through two 8-bit bi-directional ports. Each of these lines can be programmed to act as either an input or an output. In addition, four peripheral control/interrupt input lines are provided. These lines can be used to interrupt the processor or for "hand-shaking" data between the processor and a peripheral device.

- High performance replacement for Motorola/AMI/MOSTEK/Hitachi peripheral adapter.
- N channel, depletion load technology, single +5V supply.
- Completely Static and TTL compatible.
- CMOS compatible peripheral control lines.
- Fully automatic "hand-shake" allows very positive control of data transfers between processor and peripheral devices.





Basic MCS6520 Interface Diagram

SUMMARY OF MCS6520 OPERATION

See MOS TECHNOLOGY Microcomputer Hardware Manual for detailed description of MCS6520 operation.

CA1/CBI CONTROL

CRA (CRB)			
Bit 1	Bit 0	Active Transition of Input Signal*	IRQA (IRQB) Interrupt Outputs
0	0	negative	Disable--remain high
0	1	negative	Enable--goes low when bit 7 in CRA (CRB) is set by active transition of signal on CA1 (CB1)
1	0	positive	Disable--remain high
1	1	positive	Enable--as explained above

*Note: Bit 7 of CRA (CRB) will be set to a logic 1 by an active transition of the CA1 (CB1) signal. This is independent of the state of Bit 0 in CRA (CRB).

CA2/CB2 INPUT MODES

CRA (CRB)				
Bit 5	Bit 4	Bit 3	Active Transition of Input Signal*	IRQA (IRQB) Interrupt Output
0	0	0	negative	Disable--remains high
0	0	1	negative	Enable--goes low when bit 6 in CRA (CRB) is set by active transition of signal on CA2 (CB2)
0	1	0	positive	Disable--remains high
0	1	1	positive	Enable--as explained above

*Note: Bit 6 of CRA (CRB) will be set to a logic 1 by an active transition of the CA2 (CB2) signal. This is independent of the state of Bit 3 in CRA (CRB).

CA2 OUTPUT MODES

CRA				
Bit 5	Bit 4	Bit 3	Mode	Description
1	0	0	"Handshake" on Read	CA2 is set high on an active transition of the CA1 interrupt input signal and set low by a microprocessor "Read A Data" operation. This allows positive control of data transfers from the peripheral device to the microprocessor.
1	0	1	Pulse Output	CA2 goes low for one cycle after a "Read A Data" operation. This pulse can be used to signal the peripheral device that data was taken.
1	1	0	Manual Output	CA2 set low
1	1	1	Manual Output	CA2 set high

CB2 OUTPUT MODES

CRB				
Bit 5	Bit 4	Bit 3	Mode	Description
1	0	0	"Handshake" on Write	CB2 is set low on microprocessor "Write B Data" operation and is set high by an active transition of the CB1 interrupt input signal. This allows positive control of data transfers from the microprocessor to the peripheral device.
1	0	1	Pulse Output	CB2 goes low for one cycle after a microprocessor "Write B Data" operation. This can be used to signal the peripheral device that data is available.
1	1	0	Manual Output	CB2 set low
1	1	1	Manual Output	CB2 set high

MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	-0.3 to +7.0	V_{dc}
Input Voltage	V_{in}	-0.3 to +7.0	V_{dc}
Operating Temperature Range	T_A	0 to +70	°C
Storage Temperature Range	T_{stg}	-55 to +150	°C

This device contains circuitry to protect the inputs against damage due to high static voltages, however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this circuit.

STATIC D.C. CHARACTERISTICS (V_{CC} = 5.0 V ± 5%, V_{SS} = 0, T_A = 25°C unless otherwise noted)

Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage (Normal Operating Levels)	V_{IH}	+2.0	-	V_{CC}	Vdc
Input Low Voltage (Normal Operating Levels)	V_{IL}	-0.3	-	+.8	Vdc
Input Threshold Voltage	V_{IT}	0.8	-	2.0	Vdc
Input Leakage Current	I_{IN}				μAdc
V_{in} = 0 to 5.0 Vdc R/W, $\overline{Reset}$, RSØ, RS1, CSØ, CS1, $\overline{CS2}$, CA1, CB1, Φ2		-	±1.0	±2.5	
Three-State (Off State Input Current (V_{in} = 0.4 to 2.4 Vdc, V_{CC} = max) DØ-D7, PBØ-PB7, CB2	I_{TSI}	-	±2.0	±10	μAdc
Input High Current (V_{IH} = 2.4 Vdc) PAØ-PA7, CA2	I_{IH}	-100	-250	-	μAdc
Input Low Current (V_{IL} = 0.4 Vdc) PAØ-PA7, CA2	I_{IL}	-	-1.0	-1.6	mAdc
Output High Voltage (V_{CC} = min, I_{Load} = -100 μAdc)	V_{OH}	2.4	-	-	Vdc
Output Low Voltage (V_{CC} = min, I_{Load} = 1.6 mAdc)	V_{OL}	-	-	+0.4	Vdc
Output High Current (Sourcing)	I_{OH}				
(V_{OH} = 2.4 Vdc)		-100	-1000	-	μAdc
(V_O = 1.5 Vdc, the current for driving other than TTL, e.g., Darlington Base) PBØ-PB7, CB2		-1.0	-2.5	-	mAdc
Output Low Current (Sinking) (V_{OL} = 0.4 Vdc)	I_{OL}	1.6	-	-	mAdc
Output Leakage Current (Off State) $\overline{IRQA}$, $\overline{IRQB}$	I_{off}	-	1.0	10	uAdc
Power Dissipation	P_D	-	200	500	mW
Input Capacitance (V_{in} - 0, T_A = 25°C, f = 1.0 MHz)	C_{in}				pF
DØ-D7, PAØ-PA7, PBØ-PB7, CA2, CB2		-	-	10	
R/W, $\overline{Reset}$, RSØ, RS1, CSØ, CS1, CS2,		-	-	7.0	
CA1, CB1, Φ2		-	-	20	
Output Capacitance (V_{in} - 0, T_A = 25°C, f = 1.0 MHz)	C_{out}	-	-	10	pF

NOTE: Negative sign indicates outward current flow, positive indicates inward flow.

FIGURE 1 – READ TIMING CHARACTERISTICS

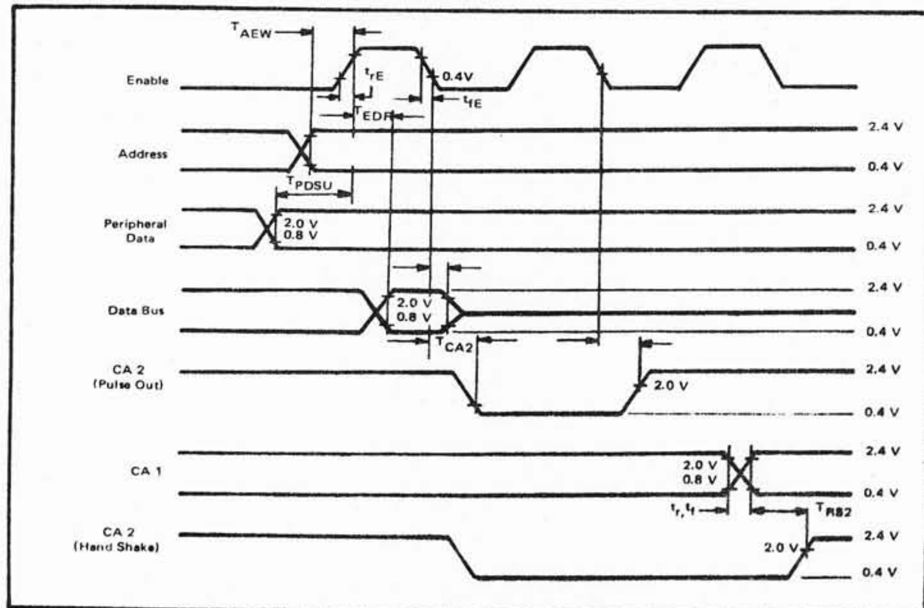
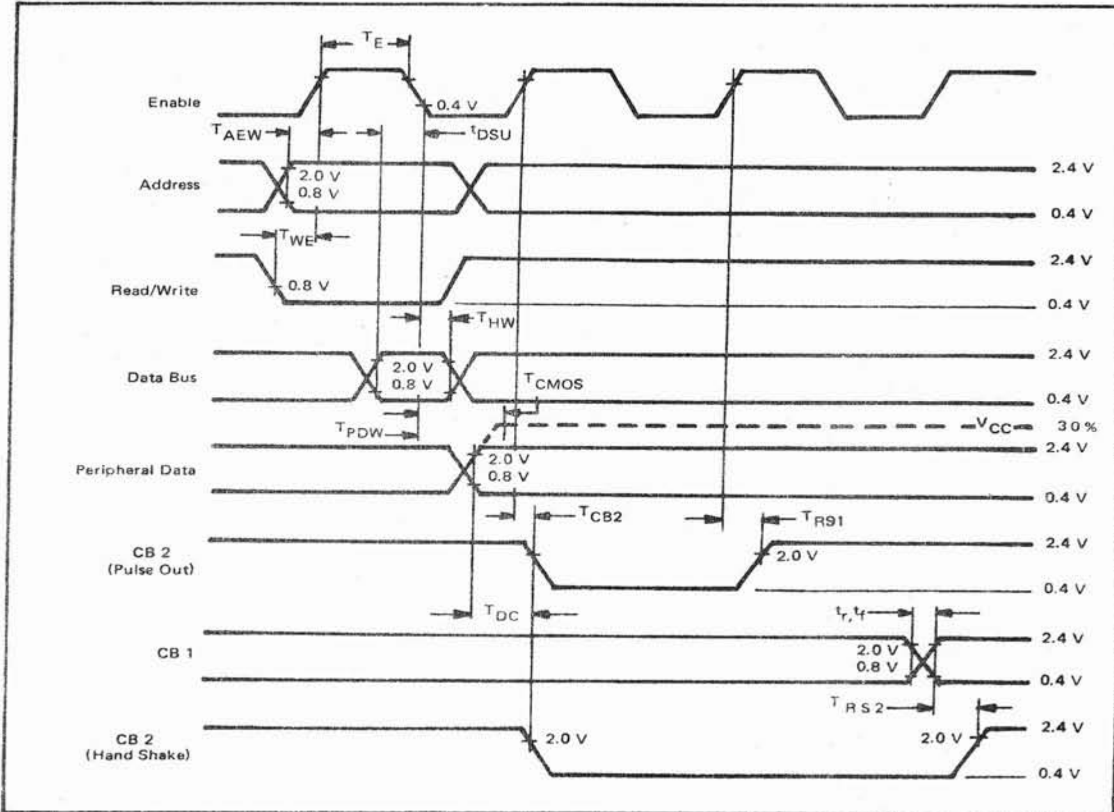


FIGURE 2 – WRITE TIMING CHARACTERISTICS



A.C. CHARACTERISTICS

Read Timing Characteristics (Figure 1, Loading 130 pF and one TTL load)

Characteristics	Symbol	Min	Typ	Max	Unit
Delay Time, Address valid to Enable positive transition	T_{AEW}	180	-	-	ns
Delay Time, Enable positive transition to Data valid on bus	T_{EDR}	-	-	395	ns
Peripheral Data Setup Time	T_{PDSU}	300	-	-	ns
Data Bus Hold Time	T_{HR}	10	-	-	ns
Delay Time, Enable negative transition to CA2 negative transition	T_{CA2}	-	-	1.0	us
Delay Time, Enable negative transition to CA2 positive transition	T_{RS1}	-	-	1.0	us
Rise and Fall Time for CA1 and CA2 input signals	t_r, t_f	-	-	1.0	us
Delay Time from CA1 active transition to CA2 positive transition	T_{RS2}	-	-	2.0	us
Rise and Fall Time for Enable input	t_{rE}, t_{fE}	-	-	25	us

Write Timing Characteristics (Figure 2)

Characteristics	Symbol	Min	Typ	Max	Unit
Enable Pulse Width	T_E	0.470	-	25	µs
Delay Time, Address valid to Enable positive transition	T_{AEW}	180	-	-	ns
Delay Time, Data valid to Enable negative transition	T_{DSU}	300	-	-	ns
Delay Time, Read/Write negative transition to Enable positive transition	T_{WE}	130	-	-	ns
Data Bus Hold Time	T_{HW}	10	-	-	ns
Delay Time, Enable negative transition to Peripheral Data valid	T_{PDW}	-	-	1.0	µs
Delay Time, Enable negative transition to Peripheral Data Valid, CMOS (V_{CC} - 30%) PAØ-PA7, CA2	T_{CMOS}	-	-	2.0	µs
Delay Time, Enable positive transition to CB2 negative transition	T_{CB2}	-	-	1.0	µs
Delay Time, Peripheral Data valid to CB2 negative transition	T_{DC}	0	-	1.5	µs
Delay Time, Enable positive transition to CB2 positive transition	T_{RS1}	-	-	1.0	µs
Rise and Fall Time for CB1 and CB2 input signals	t_r, t_f	-	-	1.0	µs
Delay Time, CB1 active transition to CB2 positive transition	T_{RS2}	-	-	2.0	µs

MOS TECHNOLOGY, INC.
VALLEY FORGE CORPORATE CENTER (215) 666-7950
950 RITTENHOUSE ROAD, NORRISTOWN, PA. 19401

PRELIMINARY
DATA
SHEET
MARCH, 1976

MCS6530 (MEMORY, I/O, TIMER ARRAY)

The MCS6530 is designed to operate in conjunction with the MCS650X Microprocessor Family. It is comprised of a mask programmable 1024 x 8 ROM, a 64 x 8 static RAM, two software controlled 8 bit bi-directional data ports allowing direct interfacing between the microprocessor unit and peripheral devices, and a software programmable interval timer with interrupt, capable of timing in various intervals from 1 to 262,144 clock periods.

* 8 bit bi-directional Data Bus for direct communication with the microprocessor
* 1024 x 8 ROM
* 64 x 8 static RAM
* Two 8 bit bi-directional data ports for interface to peripherals
* Two programmable I/O Peripheral Data Direction Registers
* Programmable Interval Timer
* Programmable Interval Timer Interrupt
* TTL & CMOS compatible peripheral lines
* Peripheral pins with Direct Transistor Drive Capability
* High Impedance Three-State Data Pins
* Allows up to 7K contiguous bytes of ROM with no external decoding

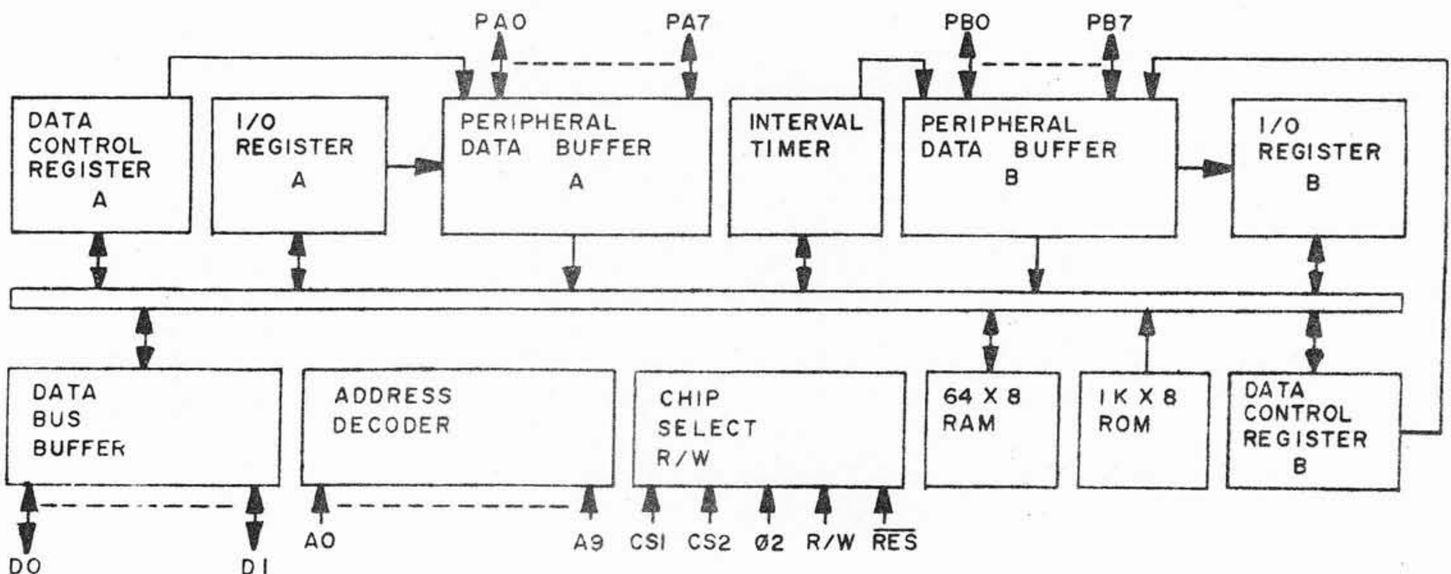


Figure 1. MCS6530 Block Diagram

MAXIMUM RATINGS

RATING	SYMBOL	VOLTAGE	UNIT
Supply Voltage	VCC	-.3 to +7.0	V
Input/Output Voltage	V_{IN}	-.3 to +7.0	V
Operating Temperature Range	T_{OP}	0 to 70	°C
Storage Temperature Range	T_{STG}	-55 to +150	°C

All inputs contain protection circuitry to prevent damage due to high static charges. Care should be exercised to prevent unnecessary application of voltage outside the specification range.

ELECTRICAL CHARACTERISTICS (VCC = 5.0v ± 5%, VSS = 0v, T_A = 25° C)

CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.	UNIT
Input High Voltage	V_{IH}	V_{SS}+2.4		VCC	V
Input Low Voltage	V_{IL}	V_{SS}-.3		V_{SS}+.4	V
Input Leakage Current; $V_{IN} = V_{SS}$ + 5v AØ-A9, RS, R/W, RES, Ø2, PB6*, PB5*	I_{IN}		1.0	2.5	µA
Input Leakage Current for High Impedance State (Three State); V_{IN} = .4v to 2.4v; DØ-D7	I_{TSI}		±1.0	±10.0	µA
Input High Current; V_{IN} = 2.4v PAØ-PA7, PBØ-PB7	I_{IH}	-100.	-300.		µA
Input Low Current; V_{IN} = .4v PAØ-PA7, PBØ-PB7	I_{IL}		-1.0	-1.6	MA
Output High Voltage VCC = MIN, $I_{LOAD} \leq$ -100µA(PAØ-PA7,PBØ-PB7,DØ-D7) $I_{LOAD} \leq$ -3 MA (PAØ,PBØ)	V_{OH}	VSS+2.4 VSS+1.5			V
Output Low Voltage VCC = MIN, $I_{LOAD} \leq$ 1.6MA	V_{OL}			VSS+.4	V
Output High Current (Sourcing); VOH ≥ 2.4v (PAØ-PA7,PBØ-PB7,DØ-D7) ≥ 1.5v Available for other than TTL (Darlingtons) (PAØ,PBØ)	I_{OH}	-100 -3.0	-1000 -5.0		µA MA
Output Low Current (Sinking); VOL ≤ .4v (PAØ-PA7) (PBØ-PB7)	I_{OL}	1.6			MA
Clock Input Capacitance	C_{Clk}			30	pf
Input Capacitance	C_{IN}			10	pf
Output Capacitance	C_{OUT}			10	pf
Power Dissipation	P_D		500	1000	MW

*When programmed as address pins
All values are D.C. readings

WRITE TIMING CHARACTERISTICS

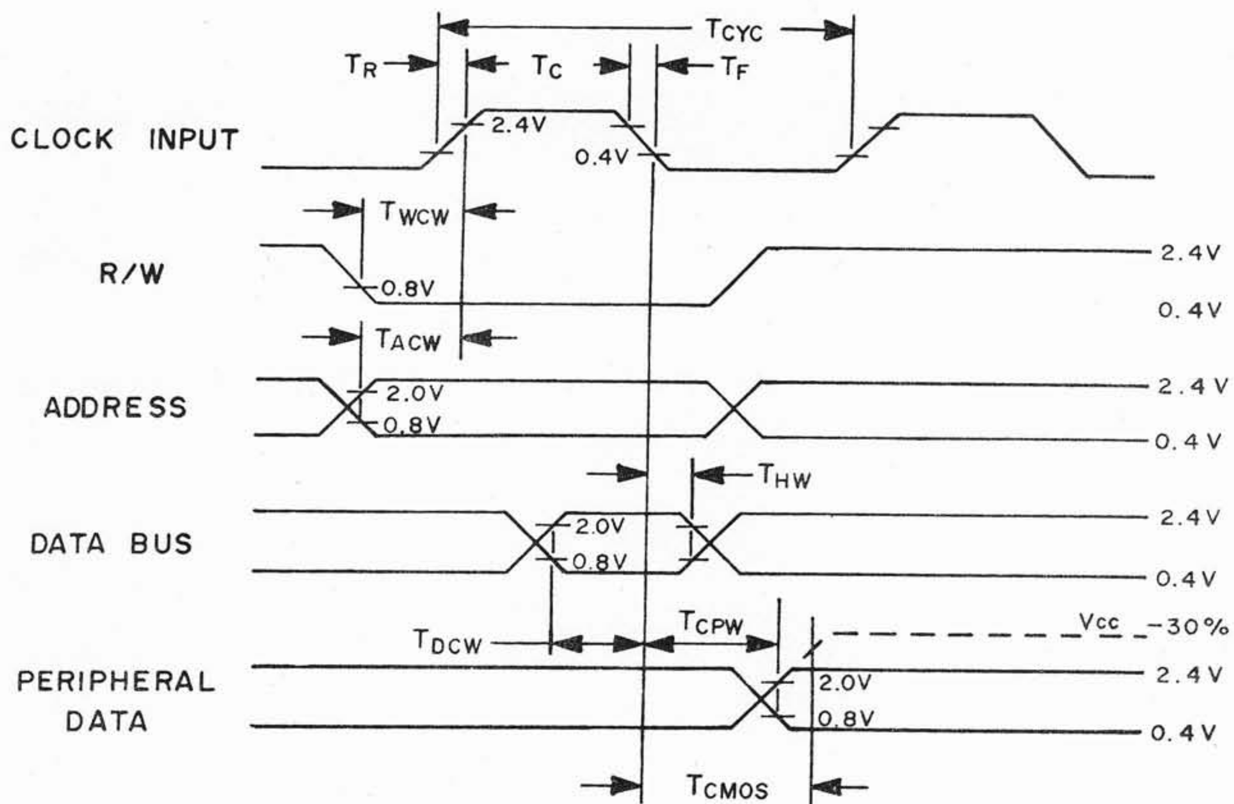
CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.	UNIT
Clock Period	T_{CYC}	1		10	μS
Rise & Fall Times	TR, TF			25	NS
Clock Pulse Width	TC	470			NS
R/W valid before positive transition of clock	TWCW	180			NS
Address valid before positive transition of clock	TACW	180			NS
Data Bus valid before negative transition of clock	TDCW	300			NS
Data Bus Hold Time	THW	10			NS
Peripheral data valid after negative transition of clock	TCPW			1	μS
Peripheral data valid after negative transition of clock driving CMOS (Level=VCC-30%)	TCMOS			2	μS

READ TIMING CHARACTERISTICS

CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.	UNIT
R/W valid before positive transition of clock	TWCR	180			NS
Address valid before positive transition of clock	TACR	180			NS
Peripheral data valid before positive transition of clock	TPCR	300			NS
Data Bus valid after positive transition of clock	TCDR			395	NS
Data Bus Hold Time	THR	10			NS
$\overline{IRQ}$ (Interval Timer Interrupt) valid before positive transition of clock	TIC	200			NS

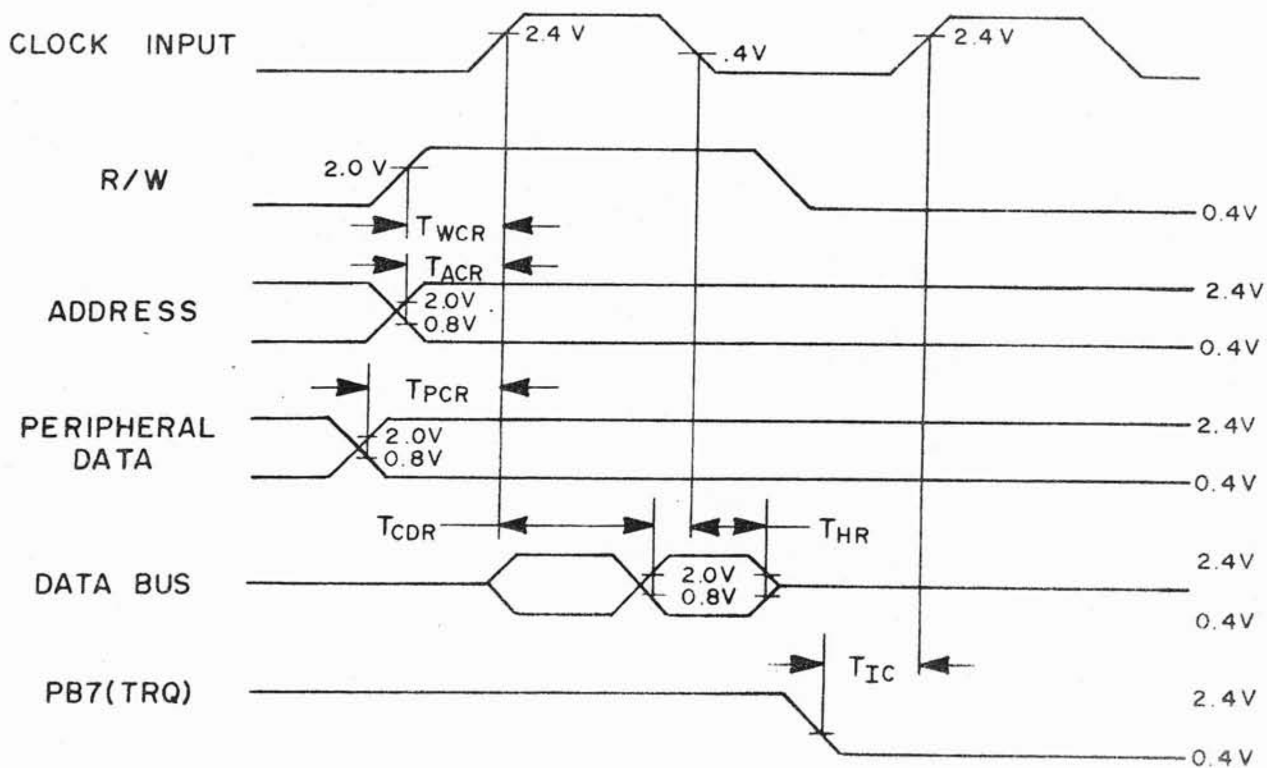
Loading = 30 pf + 1 TTL load for PAØ-PA7, PBØ-PB7

=130 pf + 1 TTL load for DØ-D7



WRITE TIMING CHARACTERISTICS

Figure 2



READ TIMING CHARACTERISTICS

Figure 3

INTERFACE SIGNAL DESCRIPTION

Reset ($\overline{RES}$)

During system initialization a Logic "0" on the $\overline{RES}$ input will cause a zeroing of all four I/O registers. This in turn will cause all I/O buses to act as inputs thus protecting external components from possible damage and erroneous data while the system is being configured under software control. The Data Bus Buffers are put into an OFF-STATE during Reset. Interrupt capability is disabled with the $\overline{RES}$ signal. The $\overline{RES}$ signal must be held low for at least one clock period when reset is required.

Input Clock

The input clock is a system Phase Two clock which can be either a low level clock ($V_{IL} < 0.4$, $V_{IH} > 2.4$) or high level clock ($V_{IL} < 0.2$, $V_{IH} = Vcc \begin{smallmatrix}+.3\\-.2\end{smallmatrix}$).

Read/Write (R/W)

The R/W signal is supplied by the microprocessor array and is used to control the transfer of data to and from the microprocessor array and the MCS6530. A high on the R/W pin allows the processor to read (with proper addressing) the data supplied by the MCS6530. A low on the R/W pin allows a write (with proper addressing) to the MCS6530.

Interrupt Request ($\overline{IRQ}$)

The $\overline{IRQ}$ pin is an interrupt pin from the interval timer. This same pin, if not used as an interrupt, can be used as a peripheral I/O pin (PB7). When used as an interrupt, the pin should be set up as an input by the data direction register. The pin will be normally high with a low indicating an interrupt from the MCS6530. An external pull-up device is not required; however, if collector-OR'd with other devices, the internal pullup may be omitted with a mask option.

Data Bus (D0-D7)

The MCS6530 has eight bi-directional data pins (D0-D7). These pins connect to the system's data lines and allow transfer of data to and from the microprocessor array. The output buffers remain in the off state except when a Read operation occurs.

Peripheral Data Ports

The MCS6530 has 16 pins available for peripheral I/O operations. Each pin is individually software programmable to act as either an input or an output. The 16 pins are divided into 2 8-bit ports, PA0-PA7 and PB0-PB7. PB5, PB6 and PB7 also have other uses which are discussed in later sections. The pins are set up as an input by writing a "0" into the corresponding bit of the data direction register. A "1" into the data direction register will cause its corresponding bit to be an output. When in the input mode, the

peripheral output buffers are in the "1" state and a pull-up device acts as less than one TTL load to the peripheral data lines. On a Read operation, the microprocessor unit reads the peripheral pin. When the peripheral device gets information from the MCS6530 it receives data stored in the data register. The microprocessor will read correct information if the peripheral lines are greater than 2.0 volts for a "1" and less than 0.8 volts for a "0" as the peripheral pins are all TTL compatible. Pins PA0 and PB0 are also capable of sourcing 3 ma at 1.5v, thus making them capable of Darlington drive.

Address Lines (A0- A9)

There are 10 address pins. In addition to these 10, there is the ROM SELECT pin. The above pins, A0-A9 and ROM SELECT, are always used as addressing pins. There are 2 additional pins which are mask programmable and can be used either individually or together as CHIP SELECTS. They are pins PB5 and PB6. When used as peripheral data pins they cannot be used as chip selects.

INTERNAL ORGANIZATION

A block diagram of the internal architecture is shown in Figure 1. The MCS6530 is divided into four basic sections, RAM, ROM, I/O and TIMER. The RAM and ROM interface directly with the microprocessor through the system data bus and address lines. The I/O section consists of 2 8-bit halves. Each half contains a Data Direction Register (DDR) and an I/O Register.

ROM 1K Byte (8K Bits)

The 8K ROM is in a 1024 x 8 configuration. Address lines A0-A9, as well as RS0 are needed to address the entire ROM. With the addition of CS1 and CS2, seven MCS6530's may be addressed, giving 7168 x 8 bits of contiguous ROM.

RAM - 64 Bytes (512 Bits)

A 64 x 8 static RAM is contained on the MCS6530. It is addressed by A0-A5 (Byte Select), RS0, A6, A7, A8, A9 and, depending on the number of chips in the system, CS1 and CS2.

Internal Peripheral Registers

There are four internal registers, two data direction registers and two peripheral I/O data registers. The two data direction registers (A side and B side) control the direction of the data into and out of the peripheral pins. A "1" written into the Data Direction Register sets up the corresponding peripheral buffer pin as an output. Therefore, anything then written into the I/O Register will appear on that corresponding peripheral

pin. A "0" written into the DDR inhibits the output buffer from transmitting data to or from the I/O Register. For example, a "1" loaded into data direction register A, position 3, sets up peripheral pin PA3 as an output. If a "0" had been loaded, PA3 would be configured as an input and remain in the high state. The two data I/O registers are used to latch data from the Data Bus during a Write operation until the peripheral device can read the data supplied by the microprocessor array.

During a read operation the microprocessor is not reading the I/O Registers but in fact is reading the peripheral data pins. For the peripheral data pins which are programmed as outputs the microprocessor will read the corresponding data bits of the I/O Register. The only way the I/O Register data can be changed is by a microprocessor Write operation. The I/O Register is not affected by a Read of the data on the peripheral pins.

Interval Timer

The Timer section of the MCS6530 contains three basic parts: preliminary divide down register, programmable 8-bit register and interrupt logic. These are illustrated in Figure 4.

The interval timer can be programmed to count up to 256 time intervals. Each time interval can be either 1T, 8T, 64T or 1024T increments, where T is the system clock period. When a full count is reached, an interrupt flag is set to a logic "1." After the interrupt flag is set the internal clock begins counting down to a maximum of -255T. Thus, after the interrupt flag is set, a Read of the timer will tell how long since the flag was set up to a maximum of 255T.

The 8 bit system Data Bus is used to transfer data to and from the Interval Timer. If a count of 52 time intervals were to be counted, the pattern 0 0 1 1 0 1 0 0 would be put on the Data Bus and written into the Interval Time register.

At the same time that data is being written to the Interval Timer, the counting intervals of 1, 8, 64, 1024T are decoded from address lines A0 and A1. During a Read or Write operation address line A3 controls the interrupt capability of PB7, i.e., $A_3 = 1$ enables IRQ on PB7, $A_3 = 0$ disables IRQ on PB7. When PB7 is to be used as an interrupt flag with the interval timer it should be programmed as an input. If PB7 is enabled by A3 and an interrupt occurs PB7 will go low. When the timer is read prior to the interrupt flag being set, the number of time intervals remaining will be read, i.e., 51, 50, 49, etc.

When the timer has counted down to 0 0 0 0 0 0 0 0 on the next count time an interrupt will occur and the counter will read 1 1 1 1 1 1 1 1 . After interrupt, the timer register decrements at a divide by "1" rate of the system clock. If after interrupt, the timer is read and a value of 1 1 1 0 0 1 0 0 is read, the time since interrupt is 28T. The value read is in two's complement.

Value read = 1 1 1 0 0 1 0 0

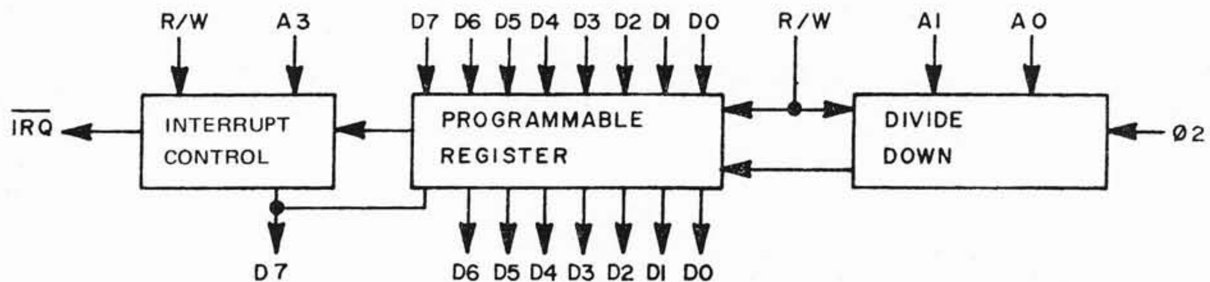
Complement = 0 0 0 1 1 0 1 1

ADD 1 = 0 0 0 1 1 1 0 0 = 28.

Thus, to arrive at the total elapsed time, merely do a two's complement add to the original time written into the timer. Again, assume time written as 0 0 1 1 0 1 0 0 (=52). With a divide by 8, total time to interrupt is (52 x 8) + 1 = 417T. Total elapsed time would be 416T + 28T = 444T, assuming the value read after interrupt was 1 1 1 0 0 1 0 0.

After the interrupt, whenever the timer is written or read the interrupt is reset. However, the reading of the timer at the same time the interrupt occurs will not reset the interrupt flag. When the interrupt flag is read on DB7 all other DB outputs (DB0 thru DB6) go to "0".

Figure 5 illustrates an example of interrupt.



BASIC ELEMENTS OF INTERVAL TIMER - Figure 4

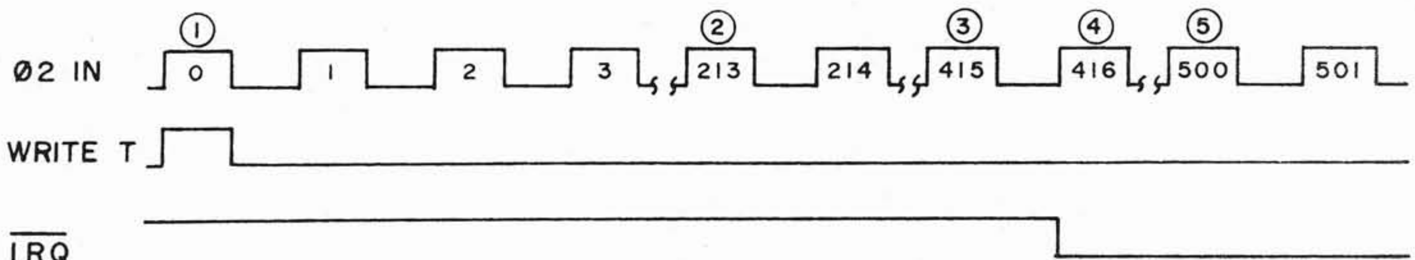


Figure 5

1. Data written into interval timer is 0 0 1 1 0 1 0 0 = 52_{10}
2. Data in Interval timer is 0 0 0 1 1 0 0 1 = 25_{10}
   $52 - \frac{213}{8} - 1 = 52-26-1 = 25$
3. Data in Interval timer is 0 0 0 0 0 0 0 0 = 0_{10}
   $52 - \frac{415}{8} - 1 = 52-51-1 = 0$
4. Interrupt has occurred at $\emptyset_2$ pulse #416
   Data in Interval timer = 1 1 1 1 1 1 1 1
5. Data in Interval timer is 1 0 1 0 1 1 0 0
   two's complement is 0 1 0 1 0 1 0 0 = 84_{10}
   $84 + (52 \times 8) = 500_{10}$

When reading the timer after an interrupt, A3 should be low so as to disable the IRQ pin. This is done so as to avoid future interrupts until after another Write timer operation.

ADDRESSING

Addressing of the MCS6530 offers many variations to the user for greater flexibility. The user may configure his system with RAM in lower memory, ROM in higher memory, and I/O registers with interval timers between the extremes. There are 10 address lines (A0-A9). In addition, there is the possibility of 3 additional address lines to be used as chip-selects and to distinguish between ROM, RAM, I/O and interval timer. Two of the additional lines are chip-selects 1 and 2 (CS1 and CS2). The chip-select pins can also be PB5 and PB6. Whether the pins are used as chip-selects or peripheral I/O pins is a mask option and must be specified when ordering the part. Both pins act independently of each other in that either or both pins may be designated as a chip-select. The third additional address line is RS0. The MCS6502 and MCS6530 in a 2-chip system would use RS0 to distinguish between ROM and non-ROM sections of the MCS6530. With the addressing pins available, a total of 7K contiguous ROM may be addressed with no external decode. Below is an example of a 1-chip and a 7-chip MCS6530 Addressing Scheme.

One-Chip Addressing

Figure 6 illustrates a 1-chip system decode for the MCS6530.

Seven-Chip Addressing

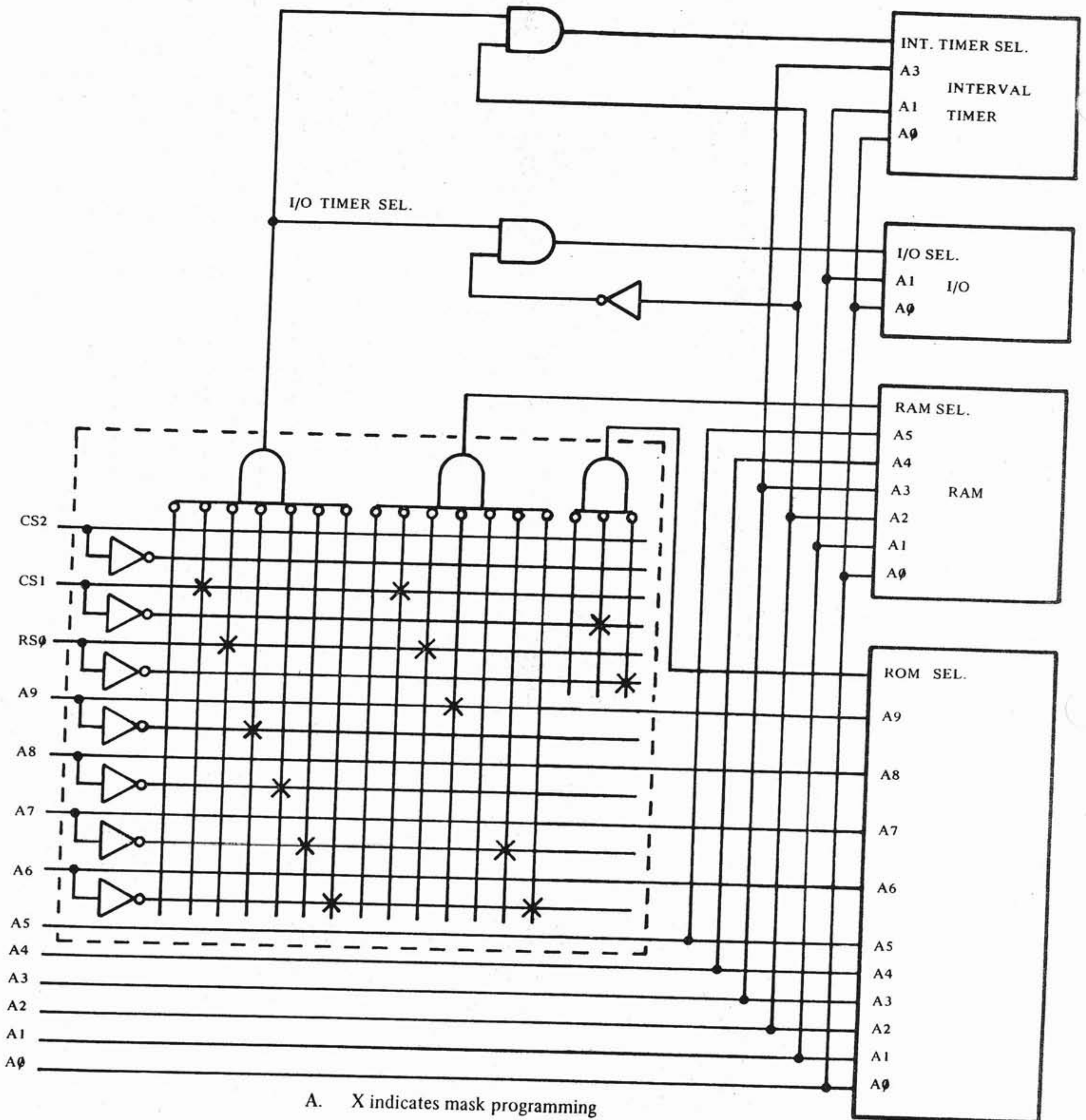
In the 7-chip system the objective would be to have 7K of contiguous ROM, with RAM in low order memory. The 7K of ROM could be placed between addresses 65,535 and 1024. For this case, assume A13, A14 and A15 are all 1 when addressing ROM, and 0 when addressing RAM or I/O. This would place the 7K ROM between Addresses 65,535 and 58,367. The 2 pins designated as chip-select or I/O would be masked programmed as chip-select pins. Pin RS0 would be connected to address line A10. Pins CS1 and CS2 would be connected to address lines A11 and A12 respectively. See Figure 7.

The two examples shown would allow addressing of the ROM and RAM; however, once the I/O or timer has been addressed, further decoding is necessary to select which of the I/O registers are desired, as well as the coding of the interval timer.

I/O Register - Timer Addressing

Figure 8 illustrates the address decoding for the internal elements and timer programming. Address lines A2 distinguishes I/O registers from the timer. When A2 is high and I/O timer select is high, the I/O registers are addressed. Once the I/O registers are addressed, address lines A1 and A0 decode the desired register.

When the timer is selected A1 and A0 decode the divide by matrix. This decoding is defined in Figure 8. In addition, Address A3 is used to enable the interrupt flag to PB7.



A. X indicates mask programming
i.e. ROM select = CS1•RSO
RAM select = $\overline{CS1}$•$\overline{RSO}$•$\overline{A9}$•A7•A6
I/O TIMER SELECT = $\overline{CS1}$•$\overline{RSO}$•A9•A8•A7•A6

B. Notice that A8 is a don't care for RAM select

C. CS2 can be used as PB5 in this example.

MCS6530 One Chip Address Encoding Diagram
Figure 6

The addressing of the ROM select, RAM select and I/O Timer select lines would be as follows:

		CS2 A12	CS1 A11	RSØ A10	A9	A8	A7	A6
MCS6530 #1,	ROM SELECT	0	0	1	X	X	X	X
	RAM SELECT	0	0	0	0	0	0	0
	I/O TIMER	0	0	0	1	0	0	0
MCS6530 #2,	ROM SELECT	0	1	0	X	X	X	X
	RAM SELECT	0	0	0	0	0	0	1
	I/O TIMER	0	0	0	1	0	0	1
MCS6530 #3,	ROM SELECT	0	1	1	X	X	X	X
	RAM SELECT	0	0	0	0	0	1	0
	I/O TIMER	0	0	0	1	0	1	0
MCS6530 #4,	ROM SELECT	1	0	0	X	X	X	X
	RAM SELECT	0	0	0	0	0	1	1
	I/O TIMER	0	0	0	1	0	1	1
MCS6530 #5,	ROM SELECT	1	0	1	X	X	X	X
	RAM SELECT	0	0	0	0	1	0	0
	I/O TIMER	0	0	0	1	1	0	0
MCS6530 #6,	ROM SELECT	1	1	0	X	X	X	X
	RAM SELECT	0	0	0	0	1	0	1
	I/O TIMER	0	0	0	1	1	0	1
MCS6530 #7,	ROM SELECT	1	1	1	X	X	X	X
	RAM SELECT	0	0	0	0	1	1	0
	I/O TIMER	0	0	0	1	1	1	0

* RAM select for MCS6530 #5 would read = $\overline{A12} \cdot \overline{A11} \cdot \overline{A10} \cdot \overline{A9} \cdot A8 \cdot \overline{A7} \cdot \overline{A6}$

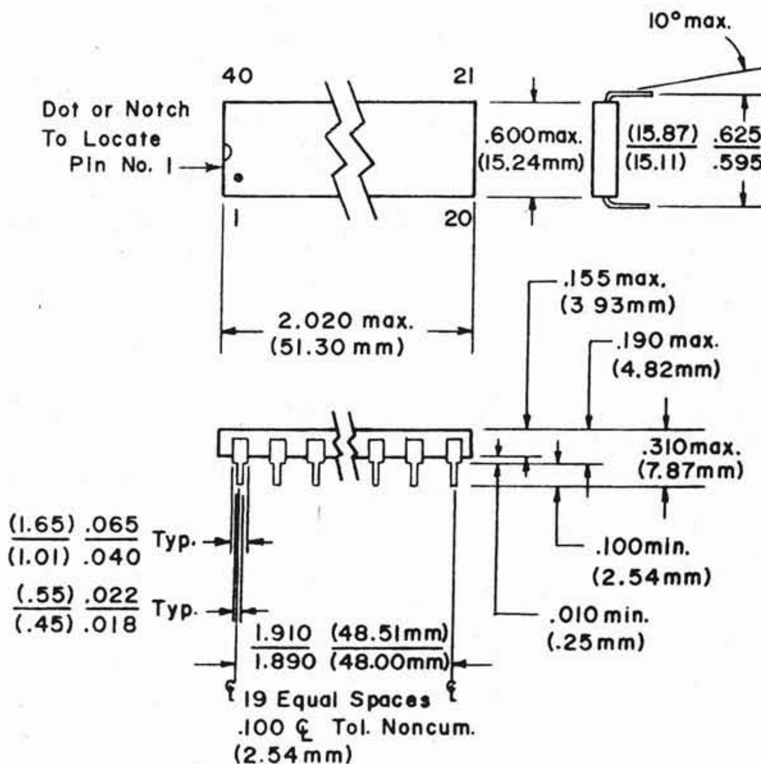
MCS6530 Seven Chip Addressing Scheme
Figure 7

ADDRESSING DECODE

	ROM SELECT	RAM SELECT	I/O TIMER SELECT	R/W	A3	A2	A1	A0
READ ROM	1	0	0	1	X	X	X	X
WRITE RAM	0	1	0	0	X	X	X	X
READ RAM	0	1	0	1	X	X	X	X
WRITE DDRA	0	0	1	0	X	0	0	1
READ DDRA	0	0	1	1	X	0	0	1
WRITE DDRB	0	0	1	0	X	0	1	1
READ DDRB	0	0	1	1	X	0	1	1
WRITE PER. REG. A	0	0	1	0	X	0	0	0
READ PER. REG. A	0	0	1	1	X	0	0	0
WRITE PER. REG. B	0	0	1	0	X	0	1	0
READ PER. REG. B	0	0	1	1	X	0	1	0
WRITE TIMER								
÷ 1T	0	0	1	0	*	1	0	0
÷ 8T	0	0	1	0	*	1	0	1
÷ 64T	0	0	1	0	*	1	1	0
÷ 1024T	0	0	1	0	*	1	1	1
READ TIMER	0	0	1	1	*	1	X	0
READ INTERRUPT FLAG	0	0	1	1	X	1	X	1

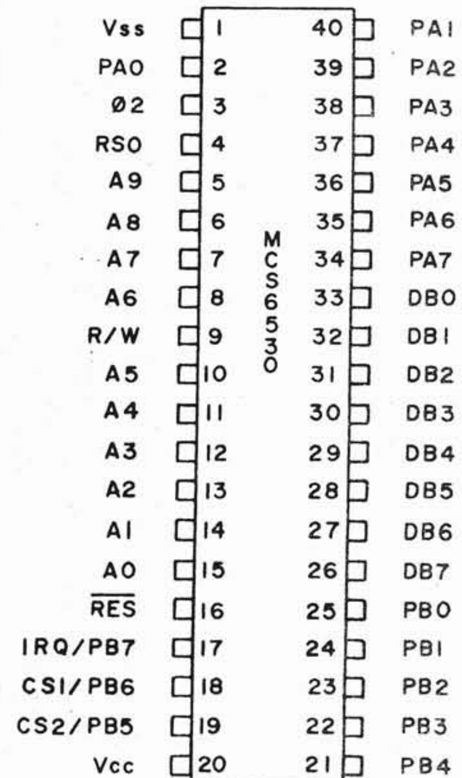
* A_3 = 1 Enables IRQ to PB7
A_3 = 0 Disables IRQ to PB7

Addressing Decode for I/O Register and Timer
FIGURE 8

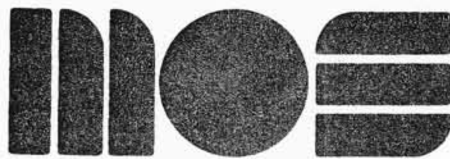


NOTE: Pin No. 1 is in lower left corner when symbolization is in normal orientation

PACKAGE OUTLINE



PIN DESIGNATION



VALLEY FORGE CORPORATE CENTER (215) 666 7950
950 RITTENHOUSE ROAD, NORRISTOWN, PA 19401

PRODUCT ANNOUNCEMENT BULLETIN

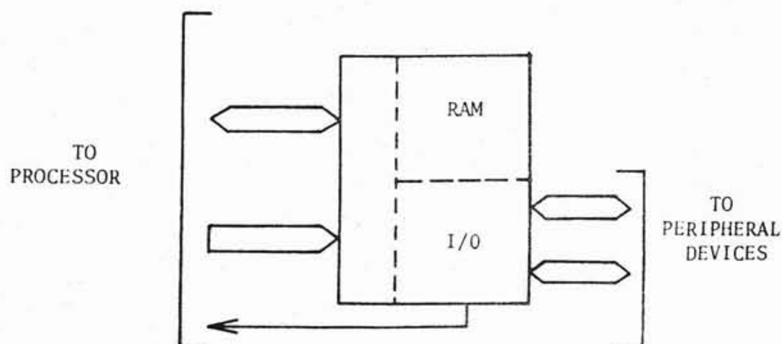
SEPTEMBER , 1976

MCS6532 RAM/IO/INTERVAL TIMER CHIP

The MCS6532 is designed to operate in conjunction with the MCS650X Microprocessor Family. It is comprised of a 128 x 8 static RAM, two software controlled 8 bit bi-directional data ports allowing direct interfacing between the microprocessor unit and peripheral devices, a software programmable interval timer with interrupt, capable of timing in various intervals from 1 to 262, 144 clock periods, and a programmable edge detect circuit.

* 8 bit bi-directional Data Bus for direct communication with the microprocessor
* Edge Sense Interrupt (Positive or Negative Edge: Programmable)
* 128 x 8 static Ram
* Two 8 bit bi-directional data ports for interface to peripherals
* Two programmable I/O Peripheral Data Direction Registers
* Programmable Interval Timer
* Programmable Interval Timer Interrupt
* TTL & CMOS compatible peripheral lines
* Peripheral pins with Direct Transistor Drive Capability
* High Impedance Three-State Data Pins

MCS6532 INTERFACE DIAGRAM



MCS6532

Signal	Pin	Pin	Signal
V_{SS}	1	40	A6
A5	2	39	$\phi 2$
A4	3	38	CS1
A3	4	37	$\overline{CS2}$
A2	5	36	$\overline{RS}$
A1	6	35	R/W
A0	7	34	$\overline{RES}$
PA0	8	33	DB0
PA1	9	32	DB1
PA2	10	31	DB2
PA3	11	30	DB3
PA4	12	29	DB4
PA5	13	28	DB5
PA6	14	27	DB6
PA7	15	26	DB7
PB7	16	25	$\overline{IRQ}$
PB6	17	24	PB0
PB5	18	23	PB1
PB4	19	22	PB2
V_{CC}	20	21	PB3

MCS6522 VERSATILE INTERFACE ADAPTER

The MOS Technology, Inc. MCS6522 is a second-generation peripheral adapter designed to bring increased capability to the microcomputer system designer for the solution of peripheral control and system timing problems. It combines the general purpose peripheral ports, handshaking, interrupt handling, etc. of the MCS6520 with a pair of very flexible interval timers and a serial-out/serial-in shift register. In addition, the chip is organized to simplify the software involved in controlling the many functions provided by this device.

Some of the important features of the MCS6522 are as follows:

* Compatible with the MCS650X and MCS651X family of microprocessors.
* Eight-bit bi-directional data bus for communication with the microprocessor.
* Two eight-bit bi-directional ports for interface to peripheral devices.
* Data Direction Registers allow each peripheral pin to act as either an input or an output.
* Interrupt Flag Register allows the microprocessor to determine the source of an interrupt very conveniently.
* Interrupt Enable Register allows very convenient control of interrupts within the chip.
* Handshake control logic for input and output peripheral data transfer operations.
* CMOS-compatible "A" and "B" peripheral ports.
* Data latching on peripheral ports.
* Two fully-programmable interval timers.
* Eight-bit Shift Register for serial interface.
* Forty-pin plastic or ceramic DIP package.

APPENDIX J

MICROCOMPUTER BIBLIOGRAPHY

BUGBOOK III — book $15.00

by Rony, Larsen, & Titus
published by K. & L. Instruments

A complete introduction to operation, programming and interfacing of an 8080 based microcomputer. Text is keyed to the use of the E. & L. MD-1 microcomputer, but is a very useful reference for all seeking hardware information about 8080 based systems.

BUGBOOK V and BUGBOOK VI — books

by Rony, Larsen, and Titus
published by E. & L. Instruments

A complete and novel treatment of microprocessors and digital circuitry. 8080 oriented but contains much useful material on interfacing microcomputers to external devices.

CMOS COOKBOOK — book $9.95

by Don Lancaster
Howard W. Sams
1977

HOW TO BUY AND USE MINICOMPUTERS & MICROCOMPUTERS — book $9.95

by William Barden, Jr.
Howard W. Sams
1976
pp. 240

INTEL 8080 ASSEMBLY LANGUAGE PROGRAMMING MANUAL — Manuf. Data $5.00

1975
pp. 75
obtain from--Intel Corp.
3065 Bowers Avenue
Santa Clara, Ca. 95051
or
Local Intel Representative or Distributor

INTEL 8080 MICROCOMPUTER SYSTEMS USER'S MANUAL — Manuf. Manual $5.00

obtain from--Intel Corp.
or
Local Intel Representative or Distributor

AN INTRODUCTION TO MICROCOMPUTERS: — books $7.50 each

by Adam Osborne
*Volume I - Basic Concepts #2001
Volume II - Some Real Products #3001
8080 Programming for Logic Design #4001
6800 Programming for Logic Design #5001
Osborne & Associates, Inc.
P.O. Box 2036
Berkeley, CA 94702

MICROCOMPUTER APPLICATIONS HANDBOOK handbook

by David J. Guzeman
Iasis Inc.
815 W. Maude Avenue
Sunnyvale, CA 94086

A complete description of hardware and software for Iasis's single borad microcomputer.

MICROCOMPUTER DESIGN book $25.00

by Donald P. Martin
1976
pp. 400
Martin Research
3336 Commercial Avenue
Northbrook, IL 60062

A comprehensive treatment of hardware and software for small microcomputer systems using the 8008 and 8080 microprocessors. This is the only book giving detailed information on the 8008.

MICROCOMPUTER AND MICROPROCESSOR book

by Hilburn and Julick
Copyright 1976 by Prentice Hall, Inc.
pp. 375

The book is intended for all persons involved in the design, use, or maintenance of digital systems using microcomputers. The book is written at a level which can be understood by persons with little previous experience.

Topics include: digital logic, number systems and codes, microcomputer architecture, software, interfacing and peripheral devices, microcomputer systems [4040, 8080, 8008, 6800, IMP-4, PPS4, COSMAC, PPS-8, PACE] design methodology and applications.

MICROPROCESSORS & MICROCOMPUTERS book $23.00

by Branko Soucek
Wiley-Interscience
1976
pp. 607

A general introduction to digital systems and microcomputers with detailed descriptions of popular 4,8,12 and 16 bit microprocessors including the 6800, 8080, and LSI-11.

MICROPROCESSORS: NEW DIRECTIONS FOR DESIGNERS — collected articles $8.95

Edited by Edward A. Torrero
1975
pp. 135

Selected articles reprinted from Electronic Design Magazine.

SCELBI SOFTWARE MANUALS — book $19.95

Machine Language Programming for the 8008
Scelbi Computer Consulting Inc.
1322 Rear Boston Post Rd.
Milford, CT 06460
pp. 170

Intro. to assembly language programming. Includes discussion of binary and floating point arithmetic.

SCELBI SOFTWARE MANUALS 8080 — books

8080 Monitor Routines	$11.95 ppd
An 8080 Assembler Program	$17.95 ppd
An 8080 Editor Program	$14.95 ppd
Scelbi Computer Consulting Inc.	all three for $39.50 ppd

Well documented software packages with program listings in Octal (paper tapes abailable). Uses non-standard memonic codes.

SCELBI'S '8080' SOFTWARE GOURMET GUIDE AND COOKBOOK — book $9.95

Machine Language Programming for the 8080
pp. 170
Scelbi Computer Consulting Inc.

Introduction to assembly language programming for the 8080. Includes several routines which can be used for number conversion floating point arithmetic and I/O processing.

SC/MP MICROPROCESSOR APPLICATIONS HANDBOOK - MANUFACTURER DATA BOOK

published by National Semiconductor Corp.
2900 Semiconductor Drive
Santa Clara, CA. 95051
available from local National Semiconductor Technical representative

Hardware and softward applications of the SC/MP.

SCELBI'S '6800' SOFTWARE GOURMET GUIDE AND COOKBOOK — book $9.95

Machine Language Programming for the 6800.
Scelbi Computer Consulting Inc.

SOFTWARD DESIGN FOR MICROPROCESSORS — book $12.95

by John G. Wester and William D. Simpson
copyright 1976 by Texas Instrument Inc.
pp. 372
order from: Texas Instruments
P.O. Box 3640, M/S-84
Dallas, TX 75285

Book was written to assist technical and non-technical people in taking their first steps toward designing with mircroprocessors and related software. Topics range from basic binary numbers to complex examples of microcomputer applications. Book was written primarily for those with little or no programming experience but it contains excellent application examples which should be of interest even to seasoned programmers.

TTL COOKBOOK — reference book $8.95

by Don Lancaster
1974
pp. 335
published by Howard W. Sams & Co., Inc.

TV TYPEWRITER COOKBOOK — reference book $8.95

by Don Lancaster
1976
published by Howard W. Sams & Company, Inc.

Mini Computers - Micro Computers.

Byte 8080 Z80

Kilobaud more Kim

Ev

B. PERIODICALS

AMERICAN LABORATORY

BYTE

published monthly by - Byte Publications, Inc.
70 Main Street
Peterborough, N.H. 03458

$12.00 per year

COMPUTER DESIGN

published monthly by - Computer Design Publishing Company
circulation address - Computer Design
Circulation Department
P.O. Box A
Winchester, MA 01890

free to qualified persons - others $20 per year

CONTROL ENGINEERING

published monthly by - Control Engineering
subscription address - 666 Fifth Avenue
New York, NY 10019

free to qualified persons

Contains useful articles on applications of microcomputers to industrial control.

DIGITAL DESIGN

published monthly by - Benwill Publishing Corp.
Circulation Director
DIGITAL DESIGN
167 Corey Road
Brookline, MA 02146

[free to qualified persons, $25.00 to others, request qualification card on company letterhead]

Dr. Dobbs Journal of COMPUTER CALISTHENICS & ORTHODONTIA

published ten (10) times per year by - Peoples Computer Company
Box E
Menlo Park, CA 94025

$12.00 per year

Devoted to publication of microcomputer oriented software such as TINY BASIC.

ELECTRONIC DESIGN

published biweekly by - Hayden Publishing Company, Inc.
50 Essex Street
Rochelle Park, NJ 07662

[free if qualified, otherwise, $30.00 per year]

ELECTRONIC DESIGN NEWS

published monthly by - Cahners Publishing Company

[free to qualified persons; very hard to get]

ELECTRONIC ENGINEERING TIMES

published biweekly by - CMP Publications
suscription address - Electronic Engineering Times
280 Community Drive
Great Neck, NY 11021

free to qualified persons

[useful for news and announcements of new microprocessor products. Has bingo card for new product ads]

ELECTRONICS

published biweekly by - McGraw-Hill, Inc.
suscription address - ELECTRONICS
McGraw-Hill Building
1221 Avenue of the Americas
New York, NY 10020

$12.00 per year to qualified persons

INTERFACE

published monthly by - Southern California Computer Society

[free with $10.00 membership in SCCS]

INTERFACE AGE [new magazine by publisher of original INTERFACE]

published monthly by - McPheters, Wolfe & Jones
6515 Sunset Blvd.
Suite 202
Hollywood, CA 90028

$10.00 per year

INSTRUMENTS & CONTROL SYSTEMS

published monthly by - Chilton Company
suscription address - Chilton Company
Chilton Way
P.O. Box 2025
Radnor, PA 19089
Attention: Circulation Dept.

[free to qualified persons, others $25.00 per year]

APPENDIX K

GLOSSARY OF COMMONLY USED TERMS

GLOSSARY OF COMMONLY USED MICROPROCESSOR TERMS

ABSOLUTE ADDRESSING - SEE DIRECT ADDRESSING

ABSOLUTE INDEXED ADDRESSING - The effective address is formed by adding the index register (X or Y) to the second and third byte of the instruction.

ACCUMULATOR - A register that holds one of the operands and the result of arithmetic and logic operations that are performed by the central processing unit. Also commonly used to hold data transferred to or from I/O devices.

ACCUMULATOR ADDRESSING - One byte instruction operating on the accumulator.

ACIA - Is an Asynchronous Communications Interface Adapter. This is an NMOS LSI device produced by Motorola for interfacing Serial ASCII devices to a micro-processor system.

ADDRESS - A number that designates a memory or I/O location.

ADDRESS BUS - A multiple-bit output Bus for transmitting an address from the CPU to the rest of the system.

ALGORITHM - The sequence of operations which defines the solution to a problem.

ALPHANUMERIC - Pertaining to a character set that contains both letters and numerals and usually other characters.

ALU (ARITHMETIC/LOCIG UNIT) - The unit of a computing system that performs arithmetic and logic operations.

ASCII CODE - The American Standard Code for Information Interchange. A seven-bit character code without the parity bit, or an eight-bit character code with the parity bit.

ASSEMBLER - A program that translates symbolic operation codes into machine language, symbolic addresses to memory addresses and assigns values to all program symbols. It translates source programs to object programs.

ASSEMBLY DIRECTIVE - A mnemonic that modifies the assembler operation but does not produce an object code (e.g., a pseudo instruction).

ASSEMBLY LANGUAGE - A collection of symbolic labels, mnemonics, and data which are to be translated into binary machine codes by the assembler.

ASYNCHRONOUS - Not occurring at the same time, or not exhibiting a constant repetition rate; irregular.

BASE - "SEE RADIX".

BCD - Binary Code Decimal. A means by which decimal numbers are represented as binary values, where integers in the range 0-9 are represented by the four-bit binary codes from 0000-1001.

BIDIRECTIONAL DATA BUS - A data bus in which digital information can be transferred in either direction.

BINARY - The base two number systems. All numbers are expressed as powers of two. As a consequence, only two symbols (0 & 1) are required to represent any number.

BIT - The smallest unit of information which can be represented. A bit may be in one of two states, represented by the binary digits 0 and 1.

BLOCK DIAGRAM - A diagram in which the essential units of any system are drawn in the form of blocks, and their relationship to each other is indicated by appropriately connected lines.

BRANCH INSTRUCTION - An instruction that causes a program jump to a specified address and execution of the instruction at that address. During the execution of the branch instruction, the central processor replaces the contents of the program counter with the specified address.

BREAKPOINT - Pertaining to a type of instruction, instruction digit, or other condition used to interrupt or stop a computer at a particular place in a program. A place in a program where such an interruption occurs or can be made to occur.

BUFFER - A noninverting digital circuit element that may be used to handle a large fan-out or to invert input and output levels.

A storage device used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transmitting data from one device to another.

BYTE - A sequence of eight adjacent binary digits operates upon as a unit.

CALL - A special type of jump in which the central processor is logically required to "remember" the contents of the program counter at the time that the jump occurs. This allows the processor later to resume execution of the main program, when it is finished with the last instruction of the subroutine.

CASCADE - An arrangement of two or more similar circuits in which the output of one circuit provides the input of the next.

CLOCK - A device or a part of a device that generates all the timing pulses for the coordination of a digital system. System clocks usually generate two or more clock phases. Each phase is a separate square wave pulse train output.

CODING - The process of preparing a program from the flow chart defining an algorithm.

COMPILER - A language translator which converts individual source statements into multiple machine instructions. A compiler translates the entire program before it is executed.

COMPLEMENT - Reverse all binary bit values (ones become zeros, zeros become ones).

CONDITIONAL - In a computer, subject to the result of a comparison made during computation.

CONDITIONAL BREAKPOINT INSTRUCTION - A conditional jump instruction that causes a computer to stop if a specified switch is set. The routine then may be allowed to proceed as coded, or a jump may be forced.

CONDITIONAL JUMP - Also called conditional transfer of control. An instruction to a computer which will cause the proper one of two (or more) addresses to be used in obtaining the next instruction, depending on some property of one or more numerical expressions or other conditions.

CONTACT BOUNCE - The uncontrolled making and breaking of a contact when the switch or relay contacts are closed. An important problem in digital circuits, where bounces can act as clock pulses.

CPU (CENTRAL PROCESSING UNIT) - The unit of a computing system that controls the interpretation and execution of instructions; inclde the ALU.

DATA BUS - A multi-line, parallel path over which digital data is transferred, from any of several destinations. Only one transfer of information can take place at any one time. While such transfer is taking place, all other sources that are tied to the bus must be disabled.

DEBUG - Detect, locate, and correct problems in a program or hardware.

DEBOUNCED - Refers to a switch or relay that no longer exhibits contact bounce.

DECODER/DRIVER - A code conversion device that can also has sufficient voltage or current output to drive an external device such as a display or a lamp monitor.

DEMULTIPLEXER - A digital device that directs information from a single input to one of several outputs. Information for output-channel selection usually is presented to the device in binary weighted form and is decoded internally. The device also acts as a single-pole multiposition switch that passes digital information in a direction opposite to that of a multiplexer.

DESTINATION - Register, memory location or I/O device which can be used to receive data during instruction execution.

DEVICE SELECT PULSE - A software-generated positive or negative clock pulse from a computer that is used to strobe the operation of one or more I/O devices, including individual integrated circuit chips.

DIRECT ADDRESSING - The second and third byte of the instruction contain the address of operand to be used.

DMA (DIRECT MEMORY ACCESS) - Suspension of processor operation to allow peripheral units external to the CPU to exercise control of memory for both READ and WRITE without altering the internal state of the processor.

DYNAMIC RAM - A random access memory that uses a capacitive element for storing a data bit. They require REFRESH.

EBCDIC - The Extended Binary Coded Decimal Interchange Code, a digital code primarily used by IBM. It closely resembles the half-ASCII code.

EDGE - The transition from logic 0 to logic 1, or from logic 1 to logic 0, in a clock pulse.

EDITOR - A program used for preparing and modifying a source program or other file by addition, deletion or change.

EFFECTIVE ADDRESS - The actual address of the desired location in memory, usually derived by some form of calculation.

EXPANSION - The process of inserting a sequence of operations represented by a macro name when the macro name is referenced in a program.

FALL TIME - The time required for an output voltage of a digital circuit to change from a logic 1 to a logic 0 state.

FAN-OUT - The number of parallel loads within a given logic family that can be driven from one output mode of a logic circuit.

FETCH - One of the two functional parts of an instruction cycle. The collective actions of acquiring a memory address, and then an instruction or data byte from memory.

FIELD - An area of an instruction mnemonic.

FILE - A collection of data records treated as a single unit.

FIFO (FIRST IN, FIRST OUT) - The term applies to the sequence of entering data into and retrieving data from data storage. The first data entered is the first data obtainable with FIFO.

FLAG - A status bit which indicates that a certain condition has arisen during the cource of arithmetic or logical manipulations or data transmission between a pair of digital electronic devices. Some flags may be tested and thus be used for determining subsequent actions.

FLAG REGISTER - A register consisting of the flag flip-flops.

FLOW CHART - A symbolic representation of the algorithm required to solve a problem.

FREQUENCY - The number of recurrences of a periodic phenomenon in a unit of time. Electrical frequency is specified as so many cycles per second, or Hertz.

FULL DUPLEX - A data transmission mode which provides simultaneous and independent transmission and reception.

HALF-ASCII - A 64-character ASCII code that contains the code words for numeric digits, alphabetic characters, and symbols but not keyboard operations.

HALF DUPLEX - A data transmission mode which provides both transmission and reception but not simultaneously.

HANDSHAKE - Interactive communication between two system components, such as between the CPU and a peripheral; often required to prevent loss of data.

HARDWARE - Physical equipment mechanical, electrical, or electronic devices.

HEXADECIMAL - A number system based upon the radix-16, in which the decimal numbers 0 through 9 and the letters A through F represent the sixteen distinct states in the code.

HIGH ADDRESS BYTE - The eight most significant bits in the 16-bit memory address word. Abbreviated H or HI.

IC (INTEGRATED CIRCUIT) - (1) A combination of interconnected circuit elements inseparably associated on or within a continuous substrate. (2) Any electronic device in which both active and passive elements are contained in a single package. In digital electronics, the term chiefly applies to circuits containing semiconductor elements.

IMMEDIATE ADDRESSING - The Operand is the second byte of the instruction, rather than its address.

IMPLIED ADDRESSING - A one-byte instruction that stipulates an operation internal to the processor. DOES NOT require any additional operand.

INCREMENT - To increase the value of a binary word. Typically, to increase the value by 1.

INDEXED ADDRESS - An indexed address is a memory address formed by adding immediate data included with the instruction to the contents of some register or memory location.

INDEXED INDIRECT ADDRESSING - The second byte of the instruction is added to the contents of the "X" index register, discarding the carry, to form a zero-page effective address.

INDIRECT ABSOLUTE ADDRESSING - The second and third bytes of the instruction contain the address for the first of two bytes in memory that contain the effective address.

INDIRECT INDEXED ADDRESSING - The second byte of this instruction is a zero-page address. The contents of this zero-page address are added to the "Y" index register to form the lower 8 bits of the effective address. Then the carry (if any) is added to the contents of the next zero-page address to form the higher 8 bits of the effective address.

INDIRECT ADDRESS - An address used with an instruction that indicates a memory location or a register that in turn contains the actual address of an operand. The indirect address may be included with the instruction, contained in a register (register indirect address) or contained in a memory location (memory directed indirect address).

INTERFACING - The joining of members of a group (such as people, instruments, etc.) in such a way that they are able to function in a compatible and coordinated fashion.

INSTRUCTION - A statement that specifies an operation and the values or locations of its operands.

INSTRUCTION CODE - A unique binary number that encodes an operation that a computer can perform.

INSTRUCTION CYCLE - A successive group of machine cycles, as few as one or as many as seven, which together perform a single microprocessor instruction within the microprocessor chip.

INSTRUCTION DECODER - A decoder within a CPU that decodes the instruction code into a series of actions that the computer performs.

INSTRUCTION REGISTER - The register that contains the instruction code.

INTERPRETER - A language translator which converts individual source statements into multiple machine instructions by translating and executing each statement as it is encountered. Can not be used to generate object code.

INTERRUPT - In a computer, a break in the normal flow of a system or routine such that the flow can be resumed from that point at a later time. The source of the interrupt may be internal or external.

I/O DEVICE - Input/output device - any digital device, including a single integrated circuit chip, that transmits data or strobe pulses to a computer or receives data or strobe pulses from a computer.

JUMP - (1) To cause the next instruction to be selected from a specified storage location in a computer. (2) A deviation from the normal sequence of execution of instructions in a computer

LABEL - One or more characters that serve to define an item of data or the location of an instruction or subroutine. A character is one symbol of a set of elementary symbols, such as those corresponding to typewriter keys.

LATCH - A simple logic storage element. A feedback loop used in a symmetrical digital circuit, such as a flip-flop, to retain a state.

LEADING EDGE - The transition of a pulse that occurs first.

LED (LIGHT-EMITTING DIODE) - A pn junction that emits light when biased in the forward direction.

LEVEL-TRIGGERED - The state of the clock input, being either logic 0 or logic 1 carries out a transfer of information or completes an action.

LIFO (LAST IN, FIRST OUT) - The latest data entered is the first data obtainable from a LIFO stach or memory section.

LSB (LEAST SIGNIFICANT BIT) - The digit with the lowest weighting in a binary number.

LISTING - An assembler output containing a listing of program mnemonics, the machine code produced, and diagnostics, if any.

LOGIC - (1) The science dealing with the basic principles and applications of truth tables, switching, gating, etc. (2) See Logical Design. (3) Also called symbolic logic. A mathematical approach to the solution of complex situations by the use of symbols to define basic concepts. The three basic logic symbols are AND, OR, and NOT. When used in Boolean algebra, these symbols are somewhat analogous to addition and multiplication. (4) In computers and information-processing networks, the systematic method that governs the operations performed on information, usually with each step influencing the one that follows. (5) The systematic plan that defines the interactions of signals in the design of a system for automatic data processing.

LOGICAL DECISION - The ability of a computer to make a choice between two alternatives; basically, the ability to answer yes or no to certain fundamental questions concerning equality and relative magnitude.

LOGICAL DESIGN - The synthesizing of a network of logical elements to perform a specified function. In digital electronics, these logical elements are digital electronic devices, such as gates, flip-flops, decoders, counters, etc.

LOGICAL ELEMENT - In a computer or data-processing system, the smallest building blocks which operators can represent in an appropriate system of symbolic logic. Typical logical elements are the AND gate and the "flip-flop".

LOOP - A sequence of instructions that is repeated until a conditional exit situation is met.

LOW ADDRESS BYTE - The eight least significant bits in the 16-bit memory address word. Abbreviated L or LO.

LSI (LARGE SCALE INTEGRATION) - Integrated circuits that perform complex functions. Such chips usually contain 100 to 2,000 gates.

MACHINE CODE - A binary code that a computer decodes to execute a specific function.

MACHINE CYCLE - A subdivision of an instruction cycle during which time a related group of actions occur within the microprocessor chip. In the 8080 microprocessor, there exist nine different machine cycles. All instructions are combinations of one or more of these machine cycles.

MACRO ASSEMBLER - An assembler routine capable of assembling programs which contain and reference macro instructions.

MACRO INSTRUCTION - A symbol that is used to represent a specified sequence of source instructions.

MAGNETIC CORE - A type of computer storage which employs a core of magnetic material with wires threaded through it. The core can be magnetized to represent a binary 1 or 0.

MAGNETIC DRUM - A storage device consisting of a rapidly rotating cylinder, the surface of which can be easily magnetized and which will retain the data. Information is stored in the form of magnetized spots (or no spots) on the drum suface.

MAGNETIC DISC - A flat circular plate with a magnetic surface on which data can be stored by selective magnetization of portions of the flat surface.

MAGNETIC TAPE - A storage system based on the use of magnetic spots (bits) on metal or coated-plastic tape. The spots are arranged so that the desired code is read out as the tape travels past the read-write head.

MASKING - A process that uses a bit pattern to select bits from a data byte for use in a subsequent operation.

MEMORY - Any device that can store logic 1 and logic 0 bits in such a manner that a single bit or group of bits can be accessed and retrieved.

MEMORY ADDRESS - A 16-bit binary number that specifies the precise memory location of a memory word among the 65,536 different possible memory locations.

MEMORY CELL - A single storage element of memory, capable of storing one bit of digital information.

MICROCOMPUTER - A computer system based on a microprocessor and contains all the memory and interface hardware necessary to perform calculations and specified information transformations.

MICROPROCESSOR - A central processing unit fabricated as one integrated circuit.

MICROPROGRAM - A computer program written in the most basic instructions or subcommands that can be executed by the computer. Frequently, it is stored in a read-only memory.

MNEMONIC - Symbols representing machine instructions designed to allow easy identification of the functions represented.

MODULO - The modulo of a counter is simply n, the number of distinct states the counter goes through before repeating. A four-bit binary counter has a modulo of 16; a decade counter has a modulo of 10; and a divide by-7 counter has a modulo of 7. In a variable modulo counter, n can be any value within a range of values.

MONITOR - Software or hardware that observes, supervises, controls, or verifies system operation.

MONOSTABLE MULTIVIBRATOR - Also called one-shot multivibrator, single-shot multi-vibrator, or start-stop multivibrator. A circuit having only one stable state, from which it can be triggered to change the state, but only for a predetermined interval, after which it returns to the original state.

MSI (MEDIUM SCALE INTEGRATION) - Integrated circuits that perform simple, self-contained logic systems, such as counters and flip-flops.

MSB (MOST SIGNIFICANT) - The digit with the highest weighting in a binary number.

MULTIPLEXER - A digital device that can select one of a number of inputs and pass the logic level of that input on to the output. Information for input-channel selection usually is presented to the device in binary weighted form and decoded internally. The device acts as a single-pole multiposition switch that passes digital information in one direction only.

NEGATIVE EDGE - The transition from logic 1 to logic 0 in a clock pulse.

NEGATIVE-EDGE TRIGGERED - Transfer of information occurs on the negative edge of the clock pulse.

NEGATIVE LOGIC - A form of logic in which the more positive voltage level represents logic 0 and the more negative level represents logic 1.

NESTING - A sequential calling of subroutines without returning to the main program.

NIBBLE - A sequence of four adjacent bits, or half a byte, is a nibble. A hexadecimal or BCD digit can be represented in a nibble.

NON-OVERLAPPING TWO-PHASE CLOCK - A two-phase clock in which the clock pulses of the individual phases do not overlap.

NON-VOLATILE MEMORY - A semiconductor memory device in which the stored digital data is not lost when the power is removed.

OCTAL - A number system based upon the radix 8, in which the decimal numbers 0 through 7 represent the eight distinct states.

ONE-BYTE-INSTRUCTION - An instruction that consists of eight contiguous bits occupying one successive location.

OPEN-COLLECTOR OUTPUT - An output from an integrated circuit device in which the final "pull-up" resistor in the output transistor for the device is missing and must be provided by the user before the circuit is completed.

OPERAND - Data which is, or will be, operated upon by an arithmetic/ logic instruction; usually identified by the address portion of an instruction, explicitly or implicitly.

OPERATION - Moving or manipulating data in the CPU or between the CPU and peripherals.

PAGE - A page consists of all the locations that can be addressed by 8-bits (a total of 256 locations) starting at 0 and going through 255. The address within a page is determined by the lower 8-bits of the address and the page number (0 through 255) is determined by the higher 8-bits of a 16-bit address.

PARITY - A method of checking the accuracy of binary numbers. If even parity is used, the sum of all the 1's in a number and its corresponding parity bit is always even. If odd parity is used, the sum of all the 1's and the parity bit is always odd.

PARTITIONING - The process of assigning specified portions of a system responsibility for performing specified functions.

PC - See "PROGRAM COUNTER"

PIA - PERIPHERAL INTERFACE ADAPTOR (MOS Technology's MPS 6520)

PERIPHERAL - A device or subsystem external to the CPU that provides additional system capabilities.

POLLING - Periodic interrogation of each of the devices that share a communications line to determine whether it requires servicing. The multiplexer or control station sends a poll that has the effect of asking the selected device, "Do you have anything to transmit?"

POP - Retrieving data from a stack.

PORT - A device or network through which data may be transferred or where device or network variables may be observed or measured.

POSITIVE EDGE - The transition from logic 0 to logic 1 in a clock pulse.

POSITIVE-EDGE TRIGGERED - Transfer of information occurs on the positive edge of the clock pulse.

POSITIVE LOGIC - A form of logic in which the more positive voltage level represents logic 1 and the more negative level represents logic 0.

PRIORITY - A preferential rating. Pertains to operations that are given preference over other system operations.

PROCESSOR - Shorthand word for microprocessor

PROGRAM - A group of instructions which causes the computer to perform a specified function.

PROGRAM COUNTER - A register containing the address of the next instruction to be executed. It is automatically incremented each time program instructions are executed.

PROGRAM LABEL - A symbol which is used to represent a memory address.

PROM (PROGRAMMABLE READ-ONLY MEMORY) - A read-only memory that is field programmable by the user.

PROPAGATION DELAY - A measure of the time required for a logic signal to travel through a logic device or a series of logic devices. It occurs as the result of four types of circuit delays - storage, rise, fall, and turn-on-delay - and is the time between when the input signal crosses the threshold - voltage point and when the responding voltage at the output crosses the same voltage point.

PSEUDO-INSTRUCTION - A mnemonic that modifies the assembler operation but does not produce an object code.

PULL-UP RESISTOR - A resistor connected to the positive supply voltage to the output collector of open-collector logic. Also used occasionally with mechanical switches to insure the voltage of one or more switch positions.

PULSE WIDTH - Also called pulse length. The time interval between the points at which the instantaneous value on the leading and trailing edges bears a specified relationship to the peak pulse amplitude.

PUSH - Putting data into a stack.

RADIX - Also called the base. The total number of distinct marks or symbols used in a numbering system. For example, since the decimal numbering system uses ten symbols, the radix is 10. In the binary numbering system, the radix is 2, because there are only two marks or symbols (0 and 1). In the octal numbering system, the radix is 8, and in the hexadecimal numbering system, the radix is 16.

RAM (RANDOM ACCESS MEMORY) - A semiconductor memory into which logic 0 and logic 1 states can be written (stored) and then read out again (retrieved).

READ - In semiconductors: To transmit data from a semiconductor memory to some other digital electronic device. The term, "read", also applies to computers and other types of memory devices.

REFRESH - The process by which dynamic RAM cells recharge the capacitive node to maintain the stored information. The charged nodes discharge due to leakage currents and without refresh, the stored data would be lost. This process must reoccur every so many microseconds. During refresh, the RAM cannot be accessed.

REFRESH LOGIC - The logic required to generate all the refresh signals and timing.

REGISTER - A hardware element used to temporarily store data.

RELATIVE ADDRESS - A relative address is a memory address formed by adding the immediate data included with the instruction to the contents of the program counter or some other register.

RESET - A computer system input that initializes and sets up certain registers in the CPU and throughout the computer system. One of the initializations, is to load a specific address into the Program Counter. The two bytes of information in that and the succeeding address is the starting address for the system program (for the MOS TECHNOLOGY processors).

RETURN - Aspecial type of jump in which the central processor resumes execution of the main program at the contents of the program counter at the time that the jump occured.

RIPPLE COUNTER - A binary counting system in which flip-flops are connected in series.

RISE TIME - The time required for an output voltage of a digital circuit to change from a logic 0 to a logic 1 state.

ROM (READ-ONLY MEMORY) - A semiconductor memory from which digital data can be repeatedly read out, but cannot be written into, as is the case for a RAM.

ROUTINE - A group of instructions that causes the computer to perform a specified function, e.g. a program.

SCRATCH PAD - The term applies to memory that is used temporarily by the CPU to store intermediate results.

SEVEN-SEGMENT DISPLAY - An electronic display that contains seven lines or segments spatially arranged in such a manner that the digits 0 through 9 can be represented through the selective lighting of certain segments to form the digit.

SEMICONDUCTOR MEMORY - A digital electronic memory device in which 1's and 0's are stored, that is a product of semiconductor manufacturing.

SHIFT REGISTER - A digital storage circuit in which information is shifted from one flip-flop of a chain to the adjacent flip-flop upon application fo each clock pulse. Data may be shifted several places to the right or left, depending on additional gating and the number of clock pulses applied to the register. Depending on the number of positions shifted, the rightmost characters are lost in a right shift, and the leftmost characters are lost in a left shift.

SIMULATOR - A program which represents the functioning of one computer system utilizing another computer system.

SOFTWARE - The means by which any defined procedure is specified for computer execution.

SOURCE - Register, memory location or I/O device which can be used to supply data for use by an instruction.

SOURCE PROGRAM - A group of statements conforming to the syntax requirements of a language processor.

SPLIT DATA BUS - Is two data buses, one for incoming communications and one for outgoing communications. An 8-bit data bus in split data bus system takes 16 lines.

STACK - A specified section of sequential memory locations used as a LIFO (Last In, First Out) file. The last element entered is the first one available for output. A stack is used to store program data, subroutine return addresses, processor status, etc.

STACK POINTER (SP) - A register which contains the address of the system read/write memory used as a stack. It is automatically incremented or decremented as instructions perform operations with the stack.

STATEMENT - An instruction in source language.

STATIC RAM - A random access memory that uses a flip-flop for storing a binary data bit. Does not require refresh.

STRING - A series of values.

SUBROUTINE - A routine that causes the execution of a specified function and which also provides for transfer of control back to the calling routine upon completion of the function.

SYMBOL - Any character string used to represent a label, mnemonic, or data constant.

SYMBOLIC ADDRESS - Also called floating address. In digital computer programming, a label chosen in a routine to identify a particular word, function, or other information that is independent of the location of the information within the routine.

SYMBOLIC CODE - A code by which programs are expressed in source language; that is, storage locations and machine operations are referred to by symbolic names and addresses that do not depend upon their hardware-determined names and addresses.

SYMBOLIC CODING - In digital computer programming, any coding system using symbolic rather than actual computer addresses.

SYNCHRONOUS - Operation of a switching network by a clock pulse generator. All circuits in the network switch simultaneously, and all actions take place synchronously with the clock.

SYNTAX ERROR - An occurrence in the source program of a label expression, or condition that does not meet the format requirements of the assembler program.

TABLE - A data structure used to contain sequences of instructions, addresses, or data constants.

TRAILING EDGE - The transition of a pulse that occurs last, such as the high-to-low transition of a positive clock pulse.

TRANSITION - The instance of changing from one state to a second state.

THREE-STATE DEVICE or TRI-STATE DEVICE - A semiconductor logic device in which there are three possible output states: (1) a "logic 0" state, (2) a "logic 1" state, or (3) a state in in which the output is, in effect, disconnected from the rest of the circuit and has no influence upon it.

THREE-BYTE INSTRUCTION - An instruction that consists of twenty-four contiguous bits occupying three successive memory locations.

TRUTH TABLE - A tabulation that shows the relation of all output logic levels of a digital circuit to all possible combinations of input logic levels in such a way as to characterize the circuit functions completely.

TWO-BYTE INSTRUCTION - An instruction that consists of sixteen contiguous bits occupying two successive memory locations.

TWO-PHASE CLOCK - A two-output timing device that provides two continuous series of timing pulse from the second series always following a single clock pulse from the first series. Depending on the type of two-phase clock, the pulses in the first and second series may or may not overlap each other. Usually identified as Phase 1 & Phase 2.

UNCONDITIONAL - Not subject to conditions external to the specific computer instruction.

UNCONDITIONAL CALL - A call instruction that is unconditional.

UNCONDITIONAL JUMP - A computer instruction that interrupts the normal process of obtaining the instructions in an ordered sequence and specifies the address from which the next instruction must be taken.

UNCONDITIONAL RETURN - A return instruction that is unconditional.

VLSI (VERY LARGE-SCALE INTEGRATION) - Monolithic digital integrated circuit chips with a typical complexity of two thousand or more gates or gate-equivalent circuits.

VOLATILE MEMORY - A semiconductor memory device in which the stored digital data is lost when the power is removed.

WEIGHTING - Most counters in the 7400 series of integrated circuit chips are weighted counters, that is, we can assign a weighted value to each of the flip-flop outputs in the counter. By summing the product of the logic state times the weighting value for each of the flip-flops, we can compute the counter state. For example, the weighting factors for a 4-bit binary counter are D = weight of 8, C = weight of 4, B = weight of 2, and A = weight of 1. The binary output, DCBA = 1101_2, from a 4-bit binary counter would therefore be 13.

WIRED-OR CIRCUIT - A circuit consisting of two or more semiconductor devices with open collector outputs in which the outputs are wired together. The output from the circuit is at a logic 0 if device A <u>or</u> device B <u>or</u> device C <u>or</u> . . . . . is at a logic 0 state.

WORD - The maximum number of binary digits that can be stored in a single addressable memory location of a given computer system.

WRITE - In semiconductors and other types of memory devices - to transmit data into a memory device from some other digital electronic device. To WRITE is to STORE.

ZERO-PAGE - The lowest 256 address locations in memory. Where the highest 8-bits of address are always 0's and the lower 8-bits identify any location from 0 to 255. Therefore, only a single byte is needed to address a location in zero-page.

ZERO-PAGE ADDRESSING - The second byte of the instruction contains a zero-page address.

ZERO-PAGE INDEXED ADDRESSING - The second byte of the instruction is added to the index register (X or Y) to form a zero-page effective address. The carry (if any) is dropped.